Package 'DeclareDesign'

August 8, 2023

Title Declare and Diagnose Research Designs

Version 1.0.4

Description Researchers can characterize and learn about the properties of research designs before implementation using `DeclareDesign`. Ex ante declaration and diagnosis of designs can help researchers clarify the strengths and limitations of their designs and to improve their properties, and can help readers evaluate a research strategy prior to implementation and without access to results. It can also make it easier for designs to be shared, replicated, and critiqued.

Depends R (>= 3.5.0), randomizr (>= 0.20.0), fabricatr (>= 0.10.0), estimatr (>= 0.20.0)

Imports rlang, generics, methods

License MIT + file LICENSE

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

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

Encoding UTF-8

RoxygenNote 7.2.3

Suggests testthat, knitr, rmarkdown, AER, diffobj, dplyr, data.table, tibble, ggplot2, future.apply, broom, MASS, Matching, betareg, biglm, gam, sf, reshape2, DesignLibrary, coin, margins, psych

NeedsCompilation no

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

Maintainer Graeme Blair <graeme.blair@gmail.com>

Repository CRAN

Date/Publication 2023-08-08 13:00:27 UTC

R topics documented:

`cite_design` *Obtain the preferred citation for a design*

Description

Obtain the preferred citation for a design

Usage

```
cite_design(design, ...)
```

Arguments

design	a design object created using the + operator
...	options for printing the citation if it is a BibTeX entry

---

`compare_diagnoses` *Compare Diagnoses*

Description

Diagnose and compare designs.

Usage

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

Arguments

design1	A design or a diagnosis.
design2	A design or a diagnosis.
sims	The number of simulations, defaulting to 1000. `sims` may also be a vector indicating the number of simulations for each step in a design, as described for `simulate_design`. Used for both designs.
bootstrap_sims	Number of bootstrap replicates for the diagnosands to obtain the standard errors of the diagnosands, defaulting to `1000`. Set to `FALSE` to turn off bootstrapping. Used for both designs. Must be greater or equal to 100.
merge_by_estimator	A logical. Whether to include `estimator` in the set of columns used for merging. Defaults to `TRUE`.
alpha	The significance level, 0.05 by default.

Details

The function `compare_diagnoses` runs a many-to-many merge matching by `inquiry` and `term` (if present). If `merge_by_estimator` equals `TRUE`, `estimator` is also included in the merging condition. Any diagnosand that is not included in both designs will be dropped from the merge.

Value

A list with a `data.frame` of compared diagnoses and both diagnoses.

Examples

```
design_a <-
  declare_model(N  = 100,
                U = rnorm(N),
                Y_Z_0 = U,
                Y_Z_1 = U + rnorm(N, mean = 2, sd = 2)) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = complete_ra(N, prob = 0.5)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

design_b <- replace_step(
  design_a, step = "assignment",
  declare_assignment(Z = complete_ra(N, prob = 0.3)) )

comparison <- compare_diagnoses(design_a, design_b, sims = 40)
```

`compare_functions` *Compare two designs*

Description

Compare two designs

Usage

```
compare_designs(
  design1,
  design2,
  format = "ansi8",
  pager = "off",
  context = -1L,
  rmd = FALSE
)

compare_design_code(
  design1,
  design2,
  format = "ansi256",
  mode = "sidebyside",
  pager = "off",
  context = -1L,
  rmd = FALSE
)

compare_design_summaries(
  design1,
```

```
  design2,
  format = "ansi256",
  mode = "sidebyside",
  pager = "off",
  context = -1L,
  rmd = FALSE
)

compare_design_data(
  design1,
  design2,
  format = "ansi256",
  mode = "sidebyside",
  pager = "off",
  context = -1L,
  rmd = FALSE
)

compare_design_estimates(
  design1,
  design2,
  format = "ansi256",
  mode = "auto",
  pager = "off",
  context = -1L,
  rmd = FALSE
)

compare_design_inquiries(
  design1,
  design2,
  format = "ansi256",
  mode = "sidebyside",
  pager = "off",
  context = -1L,
  rmd = FALSE
)
```

Arguments

design1	A design object, typically created using the + operator
design2	A design object, typically created using the + operator
format	Format (in console or HTML) options from `diffobj::diffChr`
pager	Pager option from `diffobj::diffChr`
context	Context option from `diffobj::diffChr` which sets the number of lines around differences that are printed. By default, all lines of the two objects are shown. To show only the lines that are different, set `context = 0`; to get one line around differences for context, set to 1.

rmd	Set to TRUE use in Rmarkdown HTML output. NB: will not work with LaTeX, Word, or other .Rmd outputs.
mode	Mode options from diffobj::diffChr

Examples

```
design1 <- declare_model(N = 100, u = rnorm(N), potential_outcomes(Y ~ Z + u)) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N, n = 75)) +
  declare_assignment(Z = complete_ra(N, m = 50)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

design2 <- declare_model(N = 200, U = rnorm(N),
                         potential_outcomes(Y ~ 0.5*Z + U)) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N, n = 100)) +
  declare_assignment(Z = complete_ra(N, m = 25)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, .method = lm_robust, inquiry = "ATE")

 compare_designs(design1, design2)
 compare_design_code(design1, design2)
 compare_design_summaries(design1, design2)
 compare_design_data(design1, design2)
 compare_design_estimates(design1, design2)
 compare_design_inquiries(design1, design2)
```

DeclareDesign *DeclareDesign package*

Description

The four main types of functions are to declare a step, to combine steps into designs, and to manipulate designs and designers (functions that return designs).

Design Steps

declare_model Model step

declare_inquiry Inquiry step

declare_sampling Data strategy step (sampling)

declare_assignment Data strategy step (assignment)

declare_measurement Data strategy step (measurement)

declare_estimator Answer strategy step (Estimator)

declare_test Answer strategy step (Testing function)

Design Objects

+ Add steps to create a design

`redesign` Change design parameters

`draw_data` Draw a simulated dataset

`run_design` Draw one set of inquiry values and estimates

`diagnose_design` Diagnose a design

`cite_design` Cite a design

Design Editing

`modify_design` Add, delete or replace a step

`redesign` Modify local variables within a design (advanced)

Designers

`expand_design` Generate designs from a designer

designs See also the `DesignLibrary` package for designers to use

`declare_assignment` *Declare Data Strategy: Assignment*

Description

Declare Data Strategy: Assignment

Usage

```
declare_assignment(..., handler = assignment_handler, label = NULL)

assignment_handler(data, ..., legacy = FALSE)
```

Arguments

`...`	arguments to be captured, and later passed to the handler
`handler`	a tidy-in, tidy-out function
`label`	a string describing the step
`data`	A data.frame.
`legacy`	Use the legacy randomizr functionality. This will be disabled in future; please use legacy = FALSE.

Value

A function that takes a data.frame as an argument and returns a data.frame with assignment columns appended.

Examples

```
# declare_assignment in use
## Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

run_design(design)

# Set up population to assign
model <- declare_model(
  villages = add_level(
    N = 30,
    N_households = sample(c(50:100), N, replace = TRUE)
  ),
  households = add_level(
    N = N_households,
    N_members = sample(c(1, 2, 3, 4), N,
                       prob = c(0.2, 0.3, 0.25, 0.25), replace = TRUE)
  ),
  individuals = add_level(
    N = N_members,
    age = sample(18:90, N, replace = TRUE),
    gender = rbinom(n = N, size = 1, prob = .5)
  )
)

# Assignment procedures
## Complete random assignment
design <-
  model +
  declare_assignment(Z = complete_ra(N = N, m = 1000))

head(draw_data(design))

## Cluster random assignment
design <-
  model +
  declare_assignment(Z = cluster_ra(clusters = villages,
                                    n = 15))

head(draw_data(design))
```

```
## Block and cluster random assignment
design <-
  model +
  declare_assignment(Z  = block_and_cluster_ra(
    blocks = villages,
    clusters = households,
    block_m = rep(20, 30)
  ))

head(draw_data(design))

## Block random assignment
design <-
  model +
  declare_assignment(Z = block_ra(blocks = gender, m = 100))

head(draw_data(design))

## Block random assignment using probabilities
design <-
  model +
  declare_assignment(Z = block_ra(blocks = gender,
                                  block_prob = c(1 / 3, 2 / 3)))

head(draw_data(design))

## Factorial assignment
design <-
  model +
  declare_assignment(Z1 = complete_ra(N = N, m = 100),
                     Z2 = block_ra(blocks = Z1))

head(draw_data(design))

## Assignment using functions outside of randomizr
design <-
  model +
  declare_assignment(Z = rbinom(n = N, size = 1, prob = 0.35))

head(draw_data(design))
```

---

declare_design *Declare a design*

Description

Declare a design

Usage

```
## S3 method for class 'dd'
lhs + rhs
```

Arguments

lhs	A step in a research design, beginning with a function that defines the model. Steps are evaluated sequentially. With the exception of the first step, all steps must be functions that take a `data.frame` as an argument and return a `data.frame`. Steps are declared using the `declare_` functions, i.e., `declare_model`, `declare_inquiry`, `declare_sampling`, `declare_assignment`, `declare_measurement`, `declare_estimator`, and `declare_test`.
rhs	A second step in a research design

Value

a design

Examples

```
design <-
  declare_model(
    N = 500,
    U = rnorm(N),
    potential_outcomes(Y ~ Z + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N, n = 250)) +
  declare_assignment(Z = complete_ra(N, m = 25)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

dat <- draw_data(design)
head(dat)

run_design(design)

# You may wish to have a design with only one step:

design <- declare_model(N = 500, noise = rnorm(N)) + NULL

dat <- draw_data(design)
head(dat)
```

declare_estimator *Declare estimator*

Description

Declares an estimator which generates estimates and associated statistics.

Use of declare_test is identical to use of declare_estimator. Use declare_test for hypothesis testing with no specific inquiry in mind; use declare_estimator for hypothesis testing when you can link each estimate to an inquiry. For example, declare_test could be used for a K-S test of distributional equality and declare_estimator for a difference-in-means estimate of an average treatment effect.

Usage

```
declare_estimator(
  ...,
  handler = label_estimator(method_handler),
  label = "estimator"
)

declare_estimators(
  ...,
  handler = label_estimator(method_handler),
  label = "estimator"
)

label_estimator(fn)

method_handler(
  data,
  ...,
  .method = estimatr::lm_robust,
  .summary = tidy_try,
  model,
  model_summary,
  term = FALSE
)
```

Arguments

...	arguments to be captured, and later passed to the handler
handler	a tidy-in, tidy-out function
label	a string describing the step
fn	A function that takes a data.frame as an argument and returns a data.frame with the estimates, summary statistics (i.e., standard error, p-value, and confidence interval), and a term column for labeling coefficient estimates.

data	a data.frame
.method	A method function, e.g. lm or glm. By default, the method is the lm_robust function from the estimatr package, which fits OLS regression and calculates robust and cluster-robust standard errors.
.summary	A method-in data-out function to extract coefficient estimates or method summary statistics, such as tidy or glance. By default, the DeclareDesign method summary function tidy_try is used, which first attempts to use the available tidy method for the method object sent to method, then if not attempts to summarize coefficients using the coef(summary()) and confint methods. If these do not exist for the method object, it fails.
model	Deprecated argument. Use .method instead.
model_summary	Deprecated argument. Use .summary instead.
term	Symbols or literal character vector of term that represent quantities of interest, i.e. Z. If FALSE, return the first non-intercept term; if TRUE return all term. To escape non-standard-evaluation use !!.

Details

declare_estimator is designed to handle two main ways of generating parameter estimates from data.

In declare_estimator, you can optionally provide the name of an inquiry or an objected created by declare_inquiry to connect your estimate(s) to inquiry(s).

The first is through label_estimator(method_handler), which is the default value of the handler argument. Users can use standard method functions like lm, glm, or iv_robust. The methods are summarized using the function passed to the summary argument. This will usually be a "tidier" like broom::tidy. The default summary function is tidy_try, which applies a tidy method if available, and if not, tries to make one on the fly.

An example of this approach is:

```
declare_estimator(Y ~ Z + X, .method = lm_robust, .summary = tidy, term = "Z", inquiry = "ATE")
```

The second approach is using a custom data-in, data-out function, usually first passed to label_estimator. The reason to pass the custom function to label_estimator first is to enable clean labeling and linking to inquiries.

An example of this approach is:

```
my_fun <- function(data){ with(data, median(Y[Z == 1]) - median(Y[Z == 0])) }

declare_estimator(handler = label_estimator(my_fun), inquiry = "ATE")
```

label_estimator takes a data-in-data out function to fn, and returns a data-in-data-out function that first runs the provided estimation function fn and then appends a label for the estimator and, if an inquiry is provided, a label for the inquiry.

Value

A function that accepts a data.frame as an argument and returns a data.frame containing the value of the estimator and associated statistics.

Examples

```
# Setup for examples
design <-
  declare_model(
    N = 500,
    gender = rbinom(N, 1, 0.5),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ rbinom(
      N, 1, prob = pnorm(0.2 * Z + 0.2 * gender + 0.1 * Z * gender + U)
    ))
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z))

run_design(design)

# default estimator is lm_robust with tidy summary
design_0 <-
  design +
  declare_estimator(Y ~ Z, inquiry = "ATE")

run_design(design_0)

# Linear regression using lm_robust and tidy summary
design_1 <-
  design +
  declare_estimator(
    formula = Y ~ Z,
    .method = lm_robust,
    .summary = tidy,
    term = "Z",
    inquiry = "ATE",
    label = "lm_no_controls"
  )

run_design(design_1)

# Use glance summary function to view model fit statistics
design_2 <-
  design +
  declare_estimator(.method = lm_robust,
                    formula = Y ~ Z,
                    .summary = glance)

run_design(design_2)

# Use declare_estimator to implement custom answer strategies
my_estimator <- function(data) {
  data.frame(estimate = mean(data$Y))
```

```
}

design_3 <-
  design +
  declare_inquiry(Y_bar = mean(Y)) +
  declare_estimator(handler = label_estimator(my_estimator),
                    label = "mean",
                    inquiry = "Y_bar")

run_design(design_3)

# Use `term` to select particular coefficients
design_4 <-
  design +
  declare_inquiry(difference_in_cates = mean(Y_Z_1[gender == 1] - Y_Z_0[gender == 1]) -
                    mean(Y_Z_1[gender == 0] - Y_Z_0[gender == 0])) +
  declare_estimator(Y ~ Z * gender,
                    term = "Z:gender",
                    inquiry = "difference_in_cates",
                    .method = lm_robust)

run_design(design_4)

# Use glm from base R
design_5 <-
  design +
  declare_estimator(Y ~ Z + gender,
                    family = "gaussian",
                    inquiry = "ATE",
                    .method = glm)

run_design(design_5)

# If we use logit, we'll need to estimate the average marginal effect with
# margins::margins. We wrap this up in function we'll pass to model_summary

library(margins) # for margins
library(broom) # for tidy

tidy_margins <- function(x) {
  tidy(margins(x, data = x$data), conf.int = TRUE)
}

design_6 <-
  design +
  declare_estimator(
    Y ~ Z + gender,
    .method = glm,
    family = binomial("logit"),
    .summary = tidy_margins,
    term = "Z"
  )
```

```
run_design(design_6)

# Multiple estimators for one inquiry

design_7 <-
  design +
  declare_estimator(Y ~ Z,
                    .method = lm_robust,
                    inquiry = "ATE",
                    label = "OLS") +
  declare_estimator(
    Y ~ Z + gender,
    .method = glm,
    family = binomial("logit"),
    .summary = tidy_margins,
    inquiry = "ATE",
    term = "Z",
    label = "logit"
  )

run_design(design_7)
```

`declare_inquiry` *Declare inquiry*

Description

Declares inquiries, or the inferential target of interest. Conceptually very close to "estimand" or "quantity of interest".

Usage

```
declare_inquiry(..., handler = inquiry_handler, label = "inquiry")

declare_inquiries(..., handler = inquiry_handler, label = "inquiry")

declare_estimand(...)

declare_estimands(...)

inquiry_handler(data, ..., subset = NULL, term = FALSE, label)
```

Arguments

`...`	arguments to be captured, and later passed to the handler
`handler`	a tidy-in, tidy-out function
`label`	a string describing the step
`data`	a data.frame

subset	a subset expression
term	TRUE/FALSE

Details

For the default diagnosands, the return value of the handler should have inquiry and estimand columns.

If term is TRUE, the names of ... will be returned in a term column, and inquiry will contain the step label. This can be used as an additional dimension for use in diagnosis.

Value

a function, I(), that accepts a data.frame as an argument and returns a data.frame containing the value of the inquiry, a^m.

Examples

```
# Set up a design for use in examples:
## Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_assignment(Z = complete_ra(N = N, m = 250)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z))

head(draw_data(design))

# Some common inquiries
design +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0))

design +
  declare_inquiry(difference_in_var = var(Y_Z_1) - var(Y_Z_0))

design +
  declare_inquiry(mean_Y = mean(Y))

# Inquiries among a subset
design +
  declare_inquiry(ATT = mean(Y_Z_1 - Y_Z_0),
                  subset = (Z == 1))

design +
  declare_inquiry(CATE = mean(Y_Z_1 - Y_Z_0),
                  subset = X == 1)
```

```
# equivalently
design +
  declare_inquiry(CATE = mean(Y_Z_1[X == 1] - Y_Z_0[X == 1]))

# Add inquiries to a design along with estimators that
# reference them
diff_in_variances <-
  function(data) {
    data.frame(estimate = with(data, var(Y[Z == 1]) - var(Y[Z == 0])))
  }

design_1 <-
  design +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0),
                  difference_in_var = var(Y_Z_1) - var(Y_Z_0)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z,
                    inquiry = "ATE",
                    label = "DIM") +
  declare_estimator(handler =
                      label_estimator(diff_in_variances),
                    inquiry = "difference_in_var",
                    label = "DIV")

run_design(design_1)

# Two inquiries using one estimator

design_2 <-
  design +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_inquiry(ATT = mean(Y_Z_1 - Y_Z_0), subset = (Z == 1)) +
  declare_estimator(Y ~ Z, inquiry = c("ATE", "ATT"))

run_design(design_2)

# Two inquiries using different coefficients from one estimator

design_3 <-
  design +
  declare_inquiry(intercept = mean(Y_Z_0),
                  slope = mean(Y_Z_1 - Y_Z_0)) +
  declare_estimator(
    Y ~ Z,
    .method = lm_robust,
    term = TRUE,
    inquiry = c("intercept", "slope")
  )

run_design(design_3)
```

```
# declare_inquiries usage
design_4 <- design +
  declare_inquiries(
    ATE = mean(Y_Z_1[X == 1] - Y_Z_0[X == 1]),
    CATE_X0 = mean(Y_Z_1[X == 0] - Y_Z_0[X == 0]),
    CATE_X1 = mean(Y_Z_1[X == 1] - Y_Z_0[X == 1]),
    Difference_in_CATEs = CATE_X1 - CATE_X0,
    mean_Y = mean(Y))

run_design(design_4)
```

`declare_measurement` *Declare measurement procedure*

Description

This function adds measured data columns that can be functions of unmeasured data columns.

Usage

```
declare_measurement(..., handler = measurement_handler, label = NULL)

measurement_handler(data, ...)
```

Arguments

`...`	arguments to be captured, and later passed to the handler
`handler`	a tidy-in, tidy-out function
`label`	a string describing the step
`data`	A data.frame.

Details

It is also possible to include measured variables in your declare_model call or to add variables using declare_step. However, putting latent variables in declare_model and variables-as-measured in declare_measurement helps communicate which parts of your research design are in M and which parts are in D.

Value

A function that returns a data.frame.

Examples

```
# declare_measurement in use
## Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

run_design(design)

# Reveal potential outcomes according to treatment assignment
design <-
  declare_model(N = 100,
                potential_outcomes(Y ~ rbinom(
                  N, size = 1, prob = 0.1 * Z + 0.5
                ))) +
  declare_assignment(Z = complete_ra(N, m = 50)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z))

head(draw_data(design))

# Generate observed measurement from a latent value
design <-
  declare_model(N = 100, latent = runif(N)) +
  declare_measurement(observed = rbinom(N, 1, prob = latent))

head(draw_data(design))

# Index creation
library(psych)

design <-
  declare_model(
    N = 500,
    X = rep(c(0, 1), each = N / 2),
    Y_1 = 0.2 * X + rnorm(N, sd = 0.25),
    Y_2 = 0.3 * X + 0.5 * rnorm(N, sd = 0.50),
    Y_3 = 0.1 * X + 0.4 * rnorm(N, sd = 0.75)) +
  declare_measurement(
    index = fa(
      r = cbind(Y_1, Y_2, Y_3),
      nfactors = 1,
      rotate = "varimax"
```

```
    )$scores
  )

draw_data(design)
```

`declare_model` *Declare the size and features of the population*

Description

Declare the size and features of the population

Usage

```
declare_model(..., handler = fabricate, label = NULL)
```

Arguments

`...`	arguments to be captured, and later passed to the handler
`handler`	a tidy-in, tidy-out function
`label`	a string describing the step

Value

A function that returns a data.frame.

Examples

```
# declare_model is usually used when concatenating
# design elements with `+`

## Example: Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = complete_ra(N = N, m = 250)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

# declare_model returns a function:
M <- declare_model(N = 100)
M()

# Declare a population from existing data
```

```
M <- declare_model(data = mtcars)
M()

# Resample from existing data
M <- declare_model(N = 100, data = mtcars, handler = resample_data)
M()

# Declare a model with covariates:
# observed covariates X1 and X2 and
# unobserved heterogeneity U that each affect
# outcome Y
M <- declare_model(
  N = 100,
  U = rnorm(N),
  X1 = rbinom(N, size = 1, prob = 0.5),
  X2 = X1 + rnorm(N),
  Y = 0.1 * X1 + 0.2 * X2 + 0.1 * X1 * X2 + U
)
M()

# We can draw correlated variables using draw_multivariate
M <-
declare_model(
  draw_multivariate(c(X1, X2) ~ MASS::mvrnorm(
    n = 1000,
    mu = c(0, 0),
    Sigma = matrix(c(1, 0.3, 0.3, 1), nrow = 2)
  )))
M()

# Declare potential outcomes model dependent on assignment Z
## Manually
M <-
  declare_model(N = 100,
                Y_Z_0 = rbinom(N, size = 1, prob = 0.5),
                Y_Z_1 = rbinom(N, size = 1, prob = 0.6)
  )
M()

## Using potential_outcomes
M <-
  declare_model(N = 100,
                potential_outcomes(Y ~ rbinom(N, size = 1, prob = 0.1 * Z + 0.5))
  )
M()

## we can draw from a distribution of effect sizes
M <-
  declare_model(
    N = 100,
    tau = runif(1, min = 0, max = 1),
    U = rnorm(N),
    potential_outcomes(Y ~ tau * Z + U)
```

```
  )
M()

## we can simulate treatment-by-covariate effect heterogeneity:
M <-
  declare_model(
    N = 100,
    U = rnorm(N),
    X = rbinom(N, 1, prob = 0.5),
    potential_outcomes(Y ~  0.3 * Z + 0.2*X + 0.1*Z*X + U)
  )
M()

## potential outcomes can respond to two treatments:
M <- declare_model(
  N = 6,
  U = rnorm(N),
  potential_outcomes(Y ~ Z1 + Z2 + U,
                     conditions = list(Z1 = c(0, 1), Z2 = c(0, 1))))

M()

# Declare a two-level hierarchical population
# containing varying numbers of individuals within
# households and an age variable defined at the individual
# level
M <- declare_model(
  households = add_level(
    N = 100,
    N_members = sample(c(1, 2, 3, 4), N,
                       prob = c(0.2, 0.3, 0.25, 0.25),
                       replace = TRUE)
  ),
  individuals = add_level(
    N = N_members,
    age = sample(18:90, N, replace = TRUE)
  )
)
M()

## Panel data have a more complex structure:
M <-
  declare_model(
    countries = add_level(
      N = 196,
      country_shock = rnorm(N)
    ),
    years = add_level(
      N = 100,
      time_trend = 1:N,
      year_shock = runif(N, 1, 10),
      nest = FALSE
    ),
```

```
    observation = cross_levels(
      by = join_using(countries, years),
      observation_shock = rnorm(N),
      Y = 0.01 * time_trend + country_shock + year_shock + observation_shock
    )
  )
M()

# Declare a population using a custom function
# the default handler is fabricatr::fabricate,
# but you can supply any function that returns a data.frame
my_model_function <- function(N) {
  data.frame(u = rnorm(N))
}

M <- declare_model(N = 10, handler = my_model_function)
M()
```

`declare_sampling` *Declare sampling procedure*

Description

Declare sampling procedure

Usage

```
declare_sampling(..., handler = sampling_handler, label = NULL)

sampling_handler(data, ..., legacy = FALSE)
```

Arguments

`...`	arguments to be captured, and later passed to the handler
`handler`	a tidy-in, tidy-out function
`label`	a string describing the step
`data`	A data.frame.
`legacy`	Use the legacy randomizr functionality. This will be disabled in future; please use legacy = FALSE.

Value

A sampling declaration, which is a function that takes a data.frame as an argument and returns a data.frame subsetted to sampled observations and (optionally) augmented with inclusion probabilities and other quantities.

Examples

```
# declare_sampling in use
## Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

run_design(design)

# Set up population to sample from
model <- declare_model(
  villages = add_level(
    N = 30,
    N_households = sample(c(50:100), N, replace = TRUE)
  ),
  households = add_level(
    N = N_households,
    N_members = sample(c(1, 2, 3, 4), N,
                       prob = c(0.2, 0.3, 0.25, 0.25), replace = TRUE)
  ),
  individuals = add_level(
    N = N_members,
    age = sample(18:90, N, replace = TRUE),
    gender = rbinom(n = N, size = 1, prob = .5)
  )
)

# Sampling procedures
## Complete random sampling
design <- model +
  declare_sampling(S = complete_rs(N = N, n = 1000))

head(draw_data(design))

## Cluster random sampling
design <- model +
  declare_sampling(S = cluster_rs(clusters = villages,
                                  n = 15))

head(draw_data(design))

## Strata and cluster random sampling
```

```
design <- model +
  declare_sampling(S  = strata_and_cluster_rs(
    strata = villages,
    clusters = households,
    strata_n = rep(20, 30)))

head(draw_data(design))

## Stratified random sampling
design <- model +
  declare_sampling(S = strata_rs(strata = gender, n = 100))

head(draw_data(design))
```

`declare_step` *Declare a custom step*

Description

With declare_step, you can include any function that takes data as one of its arguments and returns data in a design declaration. The first argument is always a "handler", which is the name of the data-in, data-out function. For handy data manipulations use `declare_step(fabricate, ...)`.

Usage

```
declare_step(
  ...,
  handler = function(data, ...f, ...) ...f(data, ...),
  label = NULL
)
```

Arguments

`...`	arguments to be captured, and later passed to the handler
`handler`	a tidy-in, tidy-out function
`label`	a string describing the step

Value

A function that returns a data.frame.

Examples

```
population <- declare_model(N = 5, noise = rnorm(N))
manipulate <- declare_step(fabricate, noise_squared = noise^2, zero = 0)
```

```
design <- population + manipulate
draw_data(design)
```

`declare_test` *Declare test*

Description

Declares an test which generates a test statistic and associated inferential statistics.

Use of `declare_test` is identical to use of `declare_estimator`. Use `declare_test` for hypothesis testing with no specific inquiry in mind; use `declare_estimator` for hypothesis testing when you can link each estimate to an inquiry. For example, `declare_test` could be used for a K-S test of distributional equality and `declare_estimator` for a difference-in-means estimate of an average treatment effect.

See `declare_estimator` help for an explanation of how to use `method_handler`, which is used identically in both `declare_estimator` and `declare_test`. The main difference between `declare_estimator` and `declare_test` is that `declare_test` does not link with an explicit inquiry.

Usage

```
declare_test(..., handler = label_test(method_handler), label = "test")

label_test(fn)
```

Arguments

`...`	arguments to be captured, and later passed to the handler
`handler`	a tidy-in, tidy-out function
`label`	a string describing the step
`fn`	A function that takes a data.frame as an argument and returns a data.frame with test statistics as columns.

Details

`label_test` takes a data-in-data out function to `fn`, and returns a data-in-data-out function that first runs the provided test function `fn` and then appends a label for the test.

Value

A function that accepts a data.frame as an argument and returns a data.frame containing the value of the test statistic and other inferential statistics.

See Also

See `declare_estimator` for documentation of the `method_handler` function.

Examples

```
# Balance test F test

balance_test_design <-
  declare_model(
    N = 100,
    cov1 = rnorm(N),
    cov2 = rnorm(N),
    cov3 = rnorm(N)
  ) +
  declare_assignment(Z = complete_ra(N, prob = 0.2)) +
  declare_test(Z ~ cov1 + cov2 + cov3, .method = lm_robust, .summary = glance)

## Not run:
diagnosis <- diagnose_design(
  design = balance_test_design,
  diagnosands = declare_diagnosands(
  false_positive_rate = mean(p.value <= 0.05))
)

## End(Not run)

# K-S test of distributional equality

ks_test <- function(data) {
  test <- with(data, ks.test(x = Y[Z == 1], y = Y[Z == 0]))
  data.frame(statistic = test$statistic, p.value = test$p.value)
}

distributional_equality_design <-
  declare_model(
    N = 100,
    Y_Z_1 = rnorm(N),
    Y_Z_0 = rnorm(N, sd = 1.5)
  ) +
  declare_assignment(Z = complete_ra(N, prob = 0.5)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_test(handler = label_test(ks_test), label = "ks-test")

## Not run:
diagnosis <- diagnose_design(
  design = distributional_equality_design,
  diagnosands = declare_diagnosands(power = mean(p.value <= 0.05))
)

## End(Not run)

# Thanks to Jake Bowers for this example

library(coin)
```

```
our_ttest <- function(data) {
  res <- coin::oneway_test(
    outcome ~ factor(Xclus),
    data = data,
    distribution = "asymptotic"
  )
  data.frame(p.value = pvalue(res)[[1]])
}

ttest_design <-
  declare_model(
    N = 100,
    Xclus = rbinom(n = N, size = 1, prob = 0.2),
    outcome = 3 + rnorm(N)) +
  declare_test(handler = label_test(our_ttest), label = "t-test")

## Not run:
diagnosis <- diagnose_design(
  design = ttest_design,
  diagnosands = declare_diagnosands(
    false_positive_rate = mean(p.value <= 0.05))
)

## End(Not run)
```

`diagnosand_handler` *Declare diagnosands*

Description

Declare diagnosands

Usage

```
diagnosand_handler(data, ..., subset = NULL, alpha = 0.05, label)

declare_diagnosands(..., handler = diagnosand_handler, label = NULL)
```

Arguments

`data`	A data.frame.
`...`	A set of new diagnosands.
`subset`	A subset of the simulations data frame within which to calculate diagnosands e.g. `subset = p.value < .05`.
`alpha`	Alpha significance level. Defaults to `.05`.
`label`	Label for the set of diagnosands.
`handler`	a tidy-in, tidy-out function

Details

If term is TRUE, the names of ... will be returned in a `term` column, and `inquiry` will contain the step label. This can be used as an additional dimension for use in diagnosis.

Diagnosands summarize the simulations generated by `diagnose_design` or `simulate_design`. Typically, the columns of the resulting simulations data.frame include the following variables: estimate, std.error, p.value, conf.low, conf.high, and inquiry. Many diagnosands will be a function of these variables.

Value

a function that returns a data.frame

Examples

```
# Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

## Not run:
# using built-in defaults:
diagnosis <- diagnose_design(design)
diagnosis

# You can choose your own diagnosands instead of the defaults:

my_diagnosands <-
  declare_diagnosands(median_bias = median(estimate - estimand))

## You can set diagnosands within the diagnose_design function
## using the 'diagnosands =' argument
diagnosis <- diagnose_design(design, diagnosands = my_diagnosands)
diagnosis

## You can also set diagnosands with set_diagnosands
design <- set_diagnosands(design, diagnosands = my_diagnosands)
diagnosis <- diagnose_design(design)
diagnosis

# If you do not specify diagnosands in diagnose_design,
```

```
#   the function default_diagnosands() is used,
#   which is reproduced below.

alpha <- 0.05

default_diagnosands <-
  declare_diagnosands(
    mean_estimand = mean(estimand),
    mean_estimate = mean(estimate),
    bias = mean(estimate - estimand),
    sd_estimate = sqrt(pop.var(estimate)),
    rmse = sqrt(mean((estimate - estimand) ^ 2)),
    power = mean(p.value <= alpha),
    coverage = mean(estimand <= conf.high & estimand >= conf.low)
  )

diagnose_design(
  design,
  diagnosands = default_diagnosands
)

# A longer list of potentially useful diagnosands might include:

extended_diagnosands <-
  declare_diagnosands(
    mean_estimand = mean(estimand),
    mean_estimate = mean(estimate),
    bias = mean(estimate - estimand),
    sd_estimate = sd(estimate),
    rmse = sqrt(mean((estimate - estimand) ^ 2)),
    power = mean(p.value <= alpha),
    coverage = mean(estimand <= conf.high & estimand >= conf.low),
    mean_se = mean(std.error),
    type_s_rate = mean((sign(estimate) != sign(estimand))[p.value <= alpha]),
    exaggeration_ratio = mean((estimate/estimand)[p.value <= alpha]),
    var_estimate = pop.var(estimate),
    mean_var_hat = mean(std.error^2),
    prop_pos_sig = mean(estimate > 0 & p.value <= alpha),
    mean_ci_length = mean(conf.high - conf.low)
  )

diagnose_design(
  design,
  diagnosands = extended_diagnosands
)

## End(Not run)
```

`diagnose_design` *Diagnose the design*

Description

Generates diagnosands from a design or simulations of a design.

Usage

```
diagnose_design(
  ...,
  diagnosands = NULL,
  sims = 500,
  bootstrap_sims = 100,
  make_groups = NULL,
  add_grouping_variables = NULL
)

diagnose_designs(
  ...,
  diagnosands = NULL,
  sims = 500,
  bootstrap_sims = 100,
  make_groups = NULL,
  add_grouping_variables = NULL
)

vars(...)
```

Arguments

`...`	A design or set of designs typically created using the + operator, or a `data.frame` of simulations, typically created by `simulate_design`.
`diagnosands`	A set of diagnosands created by `declare_diagnosands`. By default, these include bias, root mean-squared error, power, frequentist coverage, the mean and standard deviation of the estimate(s), the "type S" error rate (Gelman and Carlin 2014), and the mean of the inquiry(s).
`sims`	The number of simulations, defaulting to 500. sims may also be a vector indicating the number of simulations for each step in a design, as described for `simulate_design`
`bootstrap_sims`	Number of bootstrap replicates for the diagnosands to obtain the standard errors of the diagnosands, defaulting to `100`. Set to FALSE to turn off bootstrapping.
`make_groups`	Add group variables within which diagnosand values will be calculated. New variables can be created or variables already in the simulations data frame selected. Type name-value pairs within the function `vars`, i.e. `vars(significant = p.value <= 0.05)`.
`add_grouping_variables`	Deprecated. Please use make_groups instead. Variables used to generate groups of simulations for diagnosis. Added to default list: c("design", "estimand_label", "estimator", "outcome", "term")

Details

If the diagnosand function contains a `group_by` attribute, it will be used to split-apply-combine diagnosands rather than the intersecting column names.

If `sims` is named, or longer than one element, a fan-out strategy is created and used instead.

If the packages `future` and `future.apply` are installed, you can set `plan` to run multiple simulations in parallel.

Value

a list with a data frame of simulations, a data frame of diagnosands, a vector of diagnosand names, and if calculated, a data frame of bootstrap replicates.

Examples

```
# Two-arm randomized experiment
n <- 500

design <-
  declare_model(
    N = 1000,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = n)) +
  declare_assignment(Z = complete_ra(N = N, m = n/2)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

## Not run:
# Diagnose design using default diagnosands
diagnosis <- diagnose_design(design)
diagnosis

# Use tidy to produce data.frame with bootstrapped standard
# errors and confidence intervals for each diagnosand
diagnosis_df <- tidy(diagnosis)
diagnosis_df

# Use sims argument to change the number of simulations used
# to calculate diagnosands, and bootstrap_sims to change how
# many bootstraps are uses to calculate standard errors.
diagnosis <- diagnose_design(design,
                             sims = 500,
                             bootstrap_sims = 150)
tidy(diagnosis)

# Select specific diagnosands
```

```
reshape_diagnosis(diagnosis, select = "Power")

# Use your own diagnosands
my_diagnosands <-
  declare_diagnosands(median_bias = median(estimate - estimand),
                      absolute_error = mean(abs(estimate - estimand)))

diagnosis <- diagnose_design(design, diagnosands = my_diagnosands)
diagnosis

get_diagnosands(diagnosis)

get_simulations(diagnosis)

# Diagnose using an existing data frame of simulations
simulations <- simulate_design(design, sims = 500)
diagnosis   <- diagnose_design(simulations_df = simulations)
diagnosis


## End(Not run)

# If you do not specify diagnosands, the function default_diagnosands() is used,
#   which is reproduced below.

alpha <- 0.05

default_diagnosands <-
  declare_diagnosands(
    mean_estimand = mean(estimand),
    mean_estimate = mean(estimate),
    bias = mean(estimate - estimand),
    sd_estimate = sqrt(pop.var(estimate)),
    rmse = sqrt(mean((estimate - estimand) ^ 2)),
    power = mean(p.value <= alpha),
    coverage = mean(estimand <= conf.high & estimand >= conf.low)
  )

diagnose_design(
  design,
  diagnosands = default_diagnosands
)

# A longer list of useful diagnosands might include:

extended_diagnosands <-
  declare_diagnosands(
    mean_estimand = mean(estimand),
    mean_estimate = mean(estimate),
    bias = mean(estimate - estimand),
    sd_estimate = sd(estimate),
    rmse = sqrt(mean((estimate - estimand) ^ 2)),
    power = mean(p.value <= alpha),
```

```
    coverage = mean(estimand <= conf.high & estimand >= conf.low),
    mean_se = mean(std.error),
    type_s_rate = mean((sign(estimate) != sign(estimand))[p.value <= alpha]),
    exaggeration_ratio = mean((estimate/estimand)[p.value <= alpha]),
    var_estimate = pop.var(estimate),
    mean_var_hat = mean(std.error^2),
    prop_pos_sig = mean(estimate > 0 & p.value <= alpha),
    mean_ci_length = mean(conf.high - conf.low)
  )

## Not run:
diagnose_design(
  design,
  diagnosands = extended_diagnosands
)

# Adding a group for within group diagnosis:
diagnosis <- diagnose_design(design,
                             make_groups = vars(significant = p.value <= 0.05),
)
diagnosis

n <- 500
design <-
  declare_model(
    N = 1000,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ rnorm(1) * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = n)) +
  declare_assignment(Z = complete_ra(N = N, m = n/2)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

diagnosis <- diagnose_design(design,
                             make_groups =
                               vars(effect_size =
                                      cut(estimand, quantile(estimand, (0:4)/4),
                                          include.lowest = TRUE)),
)
diagnosis

# redesign can be used in conjunction with diagnose_designs
# to optimize the design for specific diagnosands
design_vary_N <- redesign(design, n = c(100, 500, 900))
diagnose_designs(design_vary_N)

# Calculate and plot the power of a design over a range of
# effect sizes
design <-
```

```
  declare_model(
    N = 200,
    U = rnorm(N),
    potential_outcomes(Y ~ runif(1, 0.0, 0.5) * Z + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = complete_ra(N)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

library(tidyverse)

simulations_df <-
  diagnose_design(design) |>
  get_simulations() |>
  mutate(significant = if_else(p.value <= 0.05, 1, 0))

ggplot(simulations_df) +
  stat_smooth(
    aes(estimand, significant),
    method = 'loess',
    color = "#3564ED",
    fill = "#72B4F3",
    formula = 'y ~ x'
  ) +
  geom_hline(
  yintercept = 0.8, color = "#C6227F", linetype = "dashed") +
  annotate("text", x = 0, y = 0.85,
    label = "Conventional power threshold = 0.8",
    hjust = 0, color = "#C6227F") +
  scale_y_continuous(breaks = seq(0, 1, 0.2)) +
  coord_cartesian(ylim = c(0, 1)) +
  theme(legend.position = "none") +
  labs(x = "Model parameter: true effect size",
       y = "Diagnosand: statistical power") +
  theme_minimal()

## End(Not run)
```

`diagnosis_helpers` *Explore your design diagnosis*

Description

Explore your design diagnosis

Usage

```
get_diagnosands(diagnosis)
```

```
get_simulations(diagnosis)
```

Arguments

diagnosis	A design diagnosis created by diagnose_design.

Examples

```
# Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

## Not run:
# Diagnose design using default diagnosands
diagnosis <- diagnose_design(design)
diagnosis

# Use get_diagnosands to explore diagnosands:
get_diagnosands(diagnosis)

# Use get_simulations to explore simulations
get_simulations(diagnosis)

# Exploring user-defined diagnosis your own diagnosands
my_diagnosands <-
  declare_diagnosands(median_bias = median(estimate - estimand),
                      absolute_error = mean(abs(estimate - estimand)))

diagnosis <- diagnose_design(design, diagnosands = my_diagnosands)
diagnosis

tidy(diagnosis)

reshape_diagnosis(diagnosis)

get_diagnosands(diagnosis)

get_simulations(diagnosis)
```

```
## End(Not run)
```

---

draw_functions *Draw data, estimates, and inquiries from a design*

Description

Draw data, estimates, and inquiries from a design

Usage

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

draw_estimand(...)

draw_estimands(...)

draw_estimates(...)
```

Arguments

design	A design object, typically created using the + operator
data	A data.frame object with sufficient information to get the data, estimates, inquiries, an assignment vector, or a sample.
start	(Defaults to 1) a scalar indicating which step in the design to begin with. By default all data steps are drawn, from step 1 to the last step of the design.
end	(Defaults to `length(design)`) a scalar indicating which step in the design to finish drawing data by.
...	A design or set of designs typically created using the + operator

Examples

```
# Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
```

```
  declare_estimator(Y ~ Z, inquiry = "ATE")

# Use draw_data to create a dataset using a design
dat <- draw_data(design)

# Use end argument to draw data up to a certain design component
dat_no_sampling <- draw_data(design, end = 3)

# Use draw_estimands to extract value of inquiry
draw_estimands(design)

# Use draw_estimates to extract value of estimator
draw_estimates(design)
```

`expand_design` *Declare a design via a designer*

Description

`expand_design` easily generates a set of design from a designer function.

Usage

```
expand_design(designer, ..., expand = TRUE, prefix = "design")
```

Arguments

`designer`	a function which yields a design
`...`	Options sent to the designer
`expand`	boolean - if true, form the crossproduct of the ..., otherwise recycle them
`prefix`	prefix for the names of the designs, i.e. if you create two designs they would be named prefix_1, prefix_2

Value

if set of designs is size one, the design, otherwise a 'by'-list of designs. Designs are given a parameters attribute with the values of parameters assigned by expand_design.

Examples

```
## Not run:

# in conjunction with DesignLibrary

library(DesignLibrary)
```

```
designs <- expand_design(multi_arm_designer, outcome_means = list(c(3,2,4), c(1,4,1)))

diagnose_design(designs)

# with a custom designer function

designer <- function(N) {
  design <-
    declare_model(
      N = N,
      U = rnorm(N),
      potential_outcomes(Y ~ 0.20 * Z + U)
    ) +
    declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
    declare_assignment(Z = complete_ra(N, m = N/2)) +
    declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
    declare_estimator(Y ~ Z, inquiry = "ATE")
  return(design)
}

# returns list of eight designs
designs <- expand_design(designer, N = seq(30, 100, 10))

# diagnose a list of designs created by expand_design or redesign
diagnosis <- diagnose_design(designs, sims = 50)

# returns a single design
large_design <- expand_design(designer, N = 200)

diagnose_large_design <- diagnose_design(large_design, sims = 50)


## End(Not run)
```

`get_functions` *Get estimates, inquiries, assignment vectors, or samples from a design given data*

Description

Get estimates, inquiries, assignment vectors, or samples from a design given data

Usage

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

Arguments

design	A design object, typically created using the + operator
data	A data.frame object with sufficient information to get the data, estimates, inquiries, an assignment vector, or a sample.
start	(Defaults to 1) a scalar indicating which step in the design to begin with. By default all data steps are drawn, from step 1 to the last step of the design.
end	(Defaults to `length(design)`) a scalar indicating which step in the design to finish with.

Examples

```
design <-
  declare_model(
    N = 100,
    U = rnorm(N),
    potential_outcomes(Y ~ Z + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N, n = 75)) +
  declare_assignment(Z = complete_ra(N, m = 50)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

dat <- draw_data(design)

draw_data(design, data = dat, start = 2)

get_estimates(design, data = dat)
```

---

modify_design *Modify a design after the fact*

Description

Insert, delete and replace steps in an (already declared) design object.

Usage

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

delete_step(design, step)

replace_step(design, step, new_step)
```

Arguments

design	A design object, usually created using the + operator, expand_design, or the design library.
new_step	The new step; Either a function or a partial call.
before	The step before which to add steps.
after	The step after which to add steps.
step	The quoted label of the step to be deleted or replaced.

Details

See modify_design for details.

Value

A new design object.

Examples

```
my_model <-
  declare_model(
    N = 100,
    U = rnorm(N),
    Y_Z_0 = U,
    Y_Z_1 = U + rnorm(N, mean = 2, sd = 2)
  )

my_assignment <- declare_assignment(Z = complete_ra(N, m = 50))
my_assignment_2 <- declare_assignment(Z = complete_ra(N, m = 25))

design <- my_model + my_assignment

draw_data(design)

design_modified <- replace_step(design, 2, my_assignment_2)

draw_data(design)

## Not run:

design <-
  declare_model(
    N = 100,
    U = rnorm(N),
    potential_outcomes(Y ~ 0.20 * Z + U)
  ) +
    declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
    declare_assignment(Z = complete_ra(N, m = N/2)) +
    declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
    declare_estimator(Y ~ Z, inquiry = "ATE")
```

```
insert_step(design, declare_sampling(S = complete_rs(N, n = 50)),
            after = 1)

# If you are using a design created by a designer, for example from
#   the DesignLibrary package, you will not have access to the step
#   objects. Instead, you can always use the label of the step.

design <- DesignLibrary::two_arm_designer()

# get the labels for the steps
names(design)

insert_step(design,
  declare_sampling(S = complete_rs(N, n = 50)),
  after = "potential_outcomes")


## End(Not run)


design <-
  declare_model(
    N = 100,
    U = rnorm(N),
    potential_outcomes(Y ~ 0.20 * Z + U)
  ) +
    declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
    declare_assignment(Z = complete_ra(N, m = N/2)) +
    declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
    declare_estimator(Y ~ Z, inquiry = "ATE")
delete_step(design, step = 5)


design <-
  declare_model(
    N = 100,
    U = rnorm(N),
    potential_outcomes(Y ~ 0.20 * Z + U)
  ) +
    declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
    declare_assignment(Z = complete_ra(N, m = N/2)) +
    declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
    declare_estimator(Y ~ Z, inquiry = "ATE")

replace_step(
  design,
  step = 3,
  new_step = declare_assignment(Z = simple_ra(N, prob = 0.5)))
```

---

pop.var *Population variance function*

Description

Population variance function

Usage

```
pop.var(x, na.rm = FALSE)
```

Arguments

x	a numeric vector, matrix or data frame.
na.rm	logical. Should missing values be removed?

Value

numeric scalar of the population variance

Examples

```
x <- 1:4
var(x) # divides by (n-1)
pop.var(x) # divides by n
```

---

post_design *Explore your design*

Description

Explore your design

Print code to recreate a design

Usage

```
print_code(design)

## S3 method for class 'design'
print(x, verbose = FALSE, ...)

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

Arguments

design	A design object, typically created using the + operator
x	a design object, typically created using the + operator
verbose	an indicator for printing a long summary of the design, defaults to TRUE
...	optional arguments to be sent to summary function
object	a design object created using the + operator

Examples

```
# Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

# Use draw_data to create a dataset using a design
dat <- draw_data(design)

draw_data(design, data = dat, start = 2)

# Apply get_estimates
get_estimates(design, data = dat)


# Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

print_code(design)
```

```
summary(design)


design <-
  declare_model(
    N = 500,
    noise = rnorm(N),
    Y_Z_0 = noise,
    Y_Z_1 = noise + rnorm(N, mean = 2, sd = 2)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N, n = 250)) +
  declare_assignment(Z = complete_ra(N, m = 25)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

summary(design)
```

`redesign` *Redesign*

Description

`redesign` quickly generates a design from an existing one by resetting symbols used in design handler parameters in a step's environment (Advanced).

Usage

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

Arguments

`design`	An object of class design.
`...`	Arguments to redesign e.g., `n = 100`. If redesigning multiple arguments, they must be specified as a named list.
`expand`	If TRUE, redesign using the crossproduct of . . . , otherwise recycle them.

Details

Warning: `redesign` will edit any symbol in your design, but if the symbol you attempt to change does not exist in a step's environment no changes will be made and no error or warning will be issued.

Please note that `redesign` functionality is experimental and may be changed in future versions.

Value

A design, or, in the case of multiple values being passed onto . . . , a 'by'-list of designs.

Examples

```
# Two-arm randomized experiment
n <- 500

design <-
  declare_model(
    N = 1000,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = n)) +
  declare_assignment(Z = complete_ra(N = N, m = n/2)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

# Use redesign to return a single modified design
modified_design <- redesign(design, n = 200)

# Use redesign to return a series of modified designs
## Sample size is varied while the rest of the design remains
## constant
design_vary_N <- redesign(design, n = c(100, 500, 900))

## Not run:
# redesign can be used in conjunction with diagnose_designs
# to optimize the design for specific diagnosands
diagnose_designs(design_vary_N)

## End(Not run)

# When redesigning with arguments that are vectors,
# use list() in redesign, with each list item
# representing a design you wish to create

prob_each <- c(.1, .5, .4)

population <- declare_model(N = 1000)
assignment <- declare_assignment(
  Z = complete_ra(prob_each = prob_each),
  legacy = FALSE)

design <- population + assignment

## returns two designs

designs_vary_prob_each <- redesign(
  design,
  prob_each = list(c(.2, .5, .3), c(0, .5, .5)))
```

```
# To illustrate what does and does not get edited by redesign,
# consider the following three designs. In the first two, argument
# X is called from the step's environment; in the third it is not.
# Using redesign will alter the role of X in the first two designs
# but not the third one.

X <- 3
f <- function(b, X) b*X
g <- function(b) b*X

design1 <- declare_model(N = 1, A = X)       + NULL
design2 <- declare_model(N = 1, A = f(2, X)) + NULL
design3 <- declare_model(N = 1, A = g(2))    + NULL

draw_data(design1)
draw_data(design2)
draw_data(design3)

draw_data(redesign(design1, X=0))
draw_data(redesign(design2, X=0))
draw_data(redesign(design3, X=0))
```

`reshape_diagnosis` *Clean up a diagnosis object for printing*

Description

Take a diagnosis object and returns a pretty output table. If diagnosands are bootstrapped, se's are put in parentheses on a second line and rounded to `digits`.

Usage

```
reshape_diagnosis(diagnosis, digits = 2, select = NULL, exclude = NULL)
```

Arguments

`diagnosis`	A diagnosis object generated by `diagnose_design`.
`digits`	Number of digits.
`select`	List of columns to include in output. Defaults to all.
`exclude`	Set of columns to exclude from output. Defaults to none.

Value

A formatted text table with bootstrapped standard errors in parentheses.

Examples

```
# Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

## Not run:
# Diagnose design using default diagnosands
diagnosis <- diagnose_design(design)
diagnosis

# Return diagnosis output table
reshape_diagnosis(diagnosis)

# Return table with subset of diagnosands
reshape_diagnosis(diagnosis, select = c("Bias", "Power"))

# With user-defined diagnosands
my_diagnosands <-
  declare_diagnosands(median_bias = median(estimate - estimand),
                      absolute_error = mean(abs(estimate - estimand)))

diagnosis <- diagnose_design(design, diagnosands = my_diagnosands)
diagnosis

reshape_diagnosis(diagnosis)

reshape_diagnosis(diagnosis, select = "Absolute Error")

# Alternative: Use tidy to produce data.frame with results of
# diagnosis including bootstrapped standard errors and
# confidence intervals for each diagnosand
diagnosis_df <- tidy(diagnosis)
diagnosis_df


## End(Not run)
```

`run_design` *Run a design one time*

Description

Run a design one time

Usage

```
run_design(design)
```

Arguments

design	a DeclareDesign object

Examples

```
# Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

# Use run_design to run a design object
run_design(design)
```

---

set_citation *Set the citation of a design*

Description

Set the citation of a design

Usage

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

Arguments

design	A design typically created using the + operator
title	The title of the design, as a character string.
author	The author(s) of the design, as a character string.
year	The year of the design, as a character string.
description	A description of the design in words, as a character string.
citation	(optional) The preferred citation for the design, as a character string, in which case title, author, year, and description may be left unspecified.

Value

a design object with a citation attribute

Examples

```
# Setup for example
design <-
  declare_model(data = sleep) +
  declare_sampling(S = complete_rs(N, n = 10))

# Set citation using set_citation
design <-
  set_citation(design,
               author = "Lovelace, Ada",
               title = "Notes",
               year = 1953,
               description =
                 "This is a text description of a design")

# View citation information using cite_design
cite_design(design)
```

---

set_diagnosands *Set the diagnosands for a design*

Description

A researcher often has a set of diagnosands in mind to appropriately assess the quality of a design. set_diagnosands sets the default diagnosands for a design, so that later readers can assess the design on the same terms as the original author. Readers can also use diagnose_design to diagnose the design using any other set of diagnosands.

Usage

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

Arguments

x	A design typically created using the + operator, or a simulations data.frame created by simulate_design.
diagnosands	A set of diagnosands created by declare_diagnosands

Value

a design object with a diagnosand attribute

Examples

```
# Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

# You can choose your own diagnosands instead of the defaults:

my_diagnosands <-
  declare_diagnosands(median_bias = median(estimate - estimand))

## Not run:
## You can set diagnosands with set_diagnosands
design <- set_diagnosands(design, diagnosands = my_diagnosands)
diagnosis <- diagnose_design(design)
diagnosis

## Using set_diagnosands to diagnose simulated data
simulations_df <- simulate_design(design)

simulations_df <- set_diagnosands(simulations_df, my_diagnosands)

diagnose_design(simulations_df)

# If you do not specify diagnosands in diagnose_design,
#   the function default_diagnosands() is used,
#   which is reproduced below.

alpha <- 0.05
```

```
default_diagnosands <-
  declare_diagnosands(
    mean_estimand = mean(estimand),
    mean_estimate = mean(estimate),
    bias = mean(estimate - estimand),
    sd_estimate = sqrt(pop.var(estimate)),
    rmse = sqrt(mean((estimate - estimand) ^ 2)),
    power = mean(p.value <= alpha),
    coverage = mean(estimand <= conf.high & estimand >= conf.low)
  )

diagnose_design(
  simulations_df,
  diagnosands = default_diagnosands
)

# A longer list of potentially useful diagnosands might include:

extended_diagnosands <-
  declare_diagnosands(
    mean_estimand = mean(estimand),
    mean_estimate = mean(estimate),
    bias = mean(estimate - estimand),
    sd_estimate = sd(estimate),
    rmse = sqrt(mean((estimate - estimand) ^ 2)),
    power = mean(p.value <= alpha),
    coverage = mean(estimand <= conf.high & estimand >= conf.low),
    mean_se = mean(std.error),
    type_s_rate = mean((sign(estimate) != sign(estimand))[p.value <= alpha]),
    exaggeration_ratio = mean((estimate/estimand)[p.value <= alpha]),
    var_estimate = pop.var(estimate),
    mean_var_hat = mean(std.error^2),
    prop_pos_sig = mean(estimate > 0 & p.value <= alpha),
    mean_ci_length = mean(conf.high - conf.low)
  )

diagnose_design(
  simulations_df,
  diagnosands = extended_diagnosands
)


## End(Not run)
```

`simulate_design` *Simulate a design*

Description

Runs many simulations of a design and returns a simulations data.frame.

Usage

```
simulate_design(..., sims = 500)

simulate_designs(..., sims = 500)
```

Arguments

...	A design created using the + operator, or a set of designs. You can also provide a single list of designs, for example one created by expand_design.
sims	The number of simulations, defaulting to 500. If sims is a vector of the form c(10, 1, 2, 1) then different steps of a design will be simulated different numbers of times.

Details

Different steps of a design may each be simulated different a number of times, as specified by sims. In this case simulations are grouped into "fans". The nested structure of simulations is recorded in the dataset using a set of variables named "step_x_draw." For example if sims = c(2,1,1,3) is passed to simulate_design, then there will be two distinct draws of step 1, indicated in variable "step_1_draw" (with values 1 and 2) and there will be three draws for step 4 within each of the step 1 draws, recorded in "step_4_draw" (with values 1 to 6).

Examples

```
# Two-arm randomized experiment
design <-
  declare_model(
    N = 500,
    gender = rbinom(N, 1, 0.5),
    X = rep(c(0, 1), each = N / 2),
    U = rnorm(N, sd = 0.25),
    potential_outcomes(Y ~ 0.2 * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_sampling(S = complete_rs(N = N, n = 200)) +
  declare_assignment(Z = complete_ra(N = N, m = 100)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

## Not run:
# Simulate design
simulations <- simulate_design(design, sims = 100)
simulations

# Diagnose design using simulations
diagnosis <- diagnose_design(simulations_df = simulations)
diagnosis

# Simulate one part of the design for a fixed population
```

```
# (The 100 simulates different assignments)
head(simulate_design(design, sims = c(1, 1, 1, 100, 1, 1)))

## End(Not run)
```

`tidy.diagnosis` *Tidy diagnosis*

Description

Tidy diagnosis

Usage

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

Arguments

`x`	A diagnosis object generated by `diagnose_design`.
`conf.int`	Logical indicating whether or not to include a confidence interval in the tidied output. Defaults to 'TRUE'.
`conf.level`	The confidence level to use for the confidence interval if 'conf.int = TRUE'. Must be strictly greater than 0 and less than 1. Defaults to 0.95, which corresponds to a 95 percent confidence interval.
`...`	extra arguments (not used)

Value

A data.frame with columns for diagnosand names, estimated diagnosand values, bootstrapped standard errors and confidence intervals

Examples

```
effect_size <- 0.1
design <-
  declare_model(
    N = 100,
    U = rnorm(N),
    X = rnorm(N),
    potential_outcomes(Y ~ effect_size * Z + X + U)
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = complete_ra(N)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE", label = "unadjusted") +
```

```
  declare_estimator(Y ~ Z + X, inquiry = "ATE", label = "adjusted")

diagnosis <- diagnose_design(design, sims = 100)

tidy(diagnosis)
```

`tidy_try` *Tidy Model Results and Filter to Relevant Coefficients*

Description

Tidy function that returns a tidy data.frame of model results and allows filtering to relevant coefficients. The function will attempt to tidy model objects even when they do not have a tidy method available. For best results, first load the broom package via `library(broom)`.

Usage

```
tidy_try(fit, term = FALSE)
```

Arguments

`fit`	A model fit, as returned by a modeling function like lm, glm, or estimatr::lm_robust.
`term`	A character vector of the terms that represent quantities of interest, i.e., "Z". If FALSE, return the first non-intercept term; if TRUE return all terms.

Value

A data.frame with coefficient estimates and associated statistics.

Examples

```
fit <- lm(mpg ~ hp + disp + cyl, data = mtcars)

tidy_try(fit)
```

Index