Package 'DeclareDesign'

January 28, 2022

Title Declare and Diagnose Research Designs

Version 0.30.0

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 7.1.2

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

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@ucla.edu>

Repository CRAN

Date/Publication 2022-01-28 06:10:02 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_assignment(Z = complete_ra(N, prob = 0.5)) +
declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
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, model = 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

```
# setting up a design stub
design <- declare_model(
  classrooms = add_level(10),
  individuals = add_level(20, female = rbinom(N, 1, 0.5))
) + NULL

# Declare assignment of m units to treatment
design + declare_assignment(Z = complete_ra(N = N, m = 100))

# Declare assignment specifying varying block probabilities
design +
  declare_assignment(Z = block_ra(blocks = female,
                                  block_prob = c(1/3, 2/3)))

# Declare assignment of clusters with probability 1/4
design + declare_assignment(
  Z = cluster_ra(prob = 1/4, clusters = classrooms))

# Declare factorial assignment (Approach 1):
#   Use complete random assignment to assign Z1
#   then block on Z1 to assign Z2.
design +
   declare_assignment(Z1 = complete_ra(N = N, m = 100),
                      Z2 = block_ra(blocks = Z1))

# Declare factorial assignment (Approach 2):
#   Assign to four conditions and then split into Z1 and Z2
design +
  declare_assignment(Z = complete_ra(N = N, conditions = 1:4),
                     Z1 = as.numeric(Z %in% 2:3),
                     Z2 = as.numeric(Z %in% 3:4))

# Declare assignment using functions outside randomizr package:
design +
  declare_assignment(Z = rbinom(n = N, size = 1, prob = 0.35))
```

`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

## Not run:
diagnosis <- diagnose_design(design)

summary(diagnosis)

## End(Not run)
```

---

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(model_handler),
  label = "estimator"
)

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

label_estimator(fn)

model_handler(
  data,
  ...,
  model = estimatr::lm_robust,
  model_summary = tidy_try,
  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
`model`	A model function, e.g. lm or glm. By default, the model is the `lm_robust` function from the estimatr package, which fits OLS regression and calculates robust and cluster-robust standard errors.
`model_summary`	A model-in data-out function to extract coefficient estimates or model summary statistics, such as `tidy` or `glance`. By default, the `DeclareDesign` model summary function `tidy_try` is used, which first attempts to use the available tidy

method for the model object sent to `model`, then if not attempts to summarize coefficients using the `coef(summary())` and `confint` methods. If these do not exist for the model object, it fails.

`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(model_handler)`, which is the default value of the `handler` argument. Users can use standard modeling functions like lm, glm, or iv_robust. The models are summarized using the function passed to the `model_summary` argument. This will usually be a "tidier" like `broom::tidy`. The default `model_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,model = lm_robust,model_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

```
# base design
design <-
  declare_model(
    N = 100,
    female = rbinom(N, 1, 0.5),
    U = rnorm(N),
    potential_outcomes(
     Y ~ rbinom(N, 1, prob = pnorm(0.2 * Z + 0.2 * female + 0.1 * Z * female + U)))
  ) +
```

```
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = complete_ra(N, m = 50)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z))

# Most estimators are modeling functions like lm or glm.

# Default statistical model is estimatr::difference_in_means
design + declare_estimator(Y ~ Z, inquiry = "ATE")

# lm from base R (classical standard errors assuming homoskedasticity)
design + declare_estimator(Y ~ Z, model = lm, inquiry = "ATE")

# Use lm_robust (linear regression with heteroskedasticity-robust standard errors)
# from `estimatr` package

design + declare_estimator(Y ~ Z, model = lm_robust, inquiry = "ATE")

# use `term` to select particular coefficients
design + declare_estimator(Y ~ Z*female, term = "Z:female", model = lm_robust)

# Use glm from base R
design + declare_estimator(
  Y ~ Z + female,
  family = "gaussian",
  inquiry = "ATE",
  model = glm
)

# 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 +
  declare_estimator(
    Y ~ Z + female,
    model = glm,
    family = binomial("logit"),
    model_summary = tidy_margins,
    term = "Z"
  )

# Multiple estimators for one inquiry

two_estimators <-
  design +
  declare_estimator(Y ~ Z,
                    model = lm_robust,
```

```
                    inquiry = "ATE",
                    label = "OLS") +
  declare_estimator(
    Y ~ Z + female,
    model = glm,
    family = binomial("logit"),
    model_summary = tidy_margins,
    inquiry = "ATE",
    term = "Z",
    label = "logit"
  )

run_design(two_estimators)

# Declare estimator using a custom handler

# Define your own estimator and use the `label_estimator` function for labeling
# Must have `data` argument that is a data.frame
my_dim_function <- function(data){
  data.frame(estimate = with(data, mean(Y[Z == 1]) - mean(Y[Z == 0])))
}

design + declare_estimator(
  handler = label_estimator(my_dim_function),
  inquiry = "ATE"
)
```

`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:

design <-
  declare_model(N = 100,
                X = rnorm(N),
                potential_outcomes(Y ~ (.25 + X) * Z + rnorm(N))) +
  declare_assignment(Z = complete_ra(N, m = 50)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z))

design + declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0))


design + declare_inquiry(ATT = mean(Y_Z_1 - Y_Z_0),
                         subset = (Z == 1))

# Add inquirys to a design along with estimators that reference them

design_1 <-
  design +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

run_design(design_1)

# Two inquirys, 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 inquirys, two 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,
    model = lm_robust,
    term = TRUE,
    inquiry = c("intercept", "slope")
  )

run_design(design_3)
```

`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

```
design <-
  declare_model(N = 6,
                U = rnorm(N),
                potential_outcomes(Y ~ Z + U)) +
  declare_assignment(Z = complete_ra(N)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z))

draw_data(design)


design <-
  declare_model(
    N = 6,
    U = rnorm(N),
    potential_outcomes(Y ~ Z1 + Z2 + U,
                       conditions = list(Z1 = c(0, 1), Z2 = c(0, 1)))) +
  declare_assignment(Z1 = complete_ra(N),
                     Z2 = block_ra(blocks = Z1),
                     legacy = FALSE) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z1 + Z2))

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 a single-level population with no covariates
declare_model(N = 100)

# declare_model returns a function:

my_model <- declare_model(N = 100)
my_model()

# Declare a single-level population with two covariates
declare_model(
  N = 6,
  female = rbinom(n = N, 1, prob = 0.5),
  height_ft = rnorm(N, mean = 5.67 - 0.33 * female, sd = 0.25)
)

# Declare a single-level population with potential outcomes
declare_model(
  N = 6,
  U = rnorm(N),
  potential_outcomes(Y ~ Z + U))


# Declare a single-level population with two sets of potential outcomes
declare_model(
  N = 6,
  U = rnorm(N),
  potential_outcomes(Y ~ Z1 + Z2 + U,
                     conditions = list(Z1 = c(0, 1), Z2 = c(0, 1))))




# Declare a population from existing data

declare_model(mtcars)

# Resample from existing data

declare_model(N = 100, data = mtcars, handler = resample_data)


# Declare a two-level hierarchical population
# containing cities within regions and a
# pollution variable defined at the city level

declare_model(regions = add_level(N = 5),
```

```
              cities = add_level(N = 10, pollution = rnorm(N, mean = 5)))

# 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))
}

declare_model(N = 10, handler = my_model_function)
```

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

Description

Deprecated. Please use declare_model instead.

Usage

```
declare_population(..., 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 potential outcomes declaration, which is a function that returns a data.frame.

Examples

```
# Declare a single-level population with no covariates
my_population <- declare_population(N = 100)

# Declare a population from existing data
my_population <- declare_population(sleep)

# Declare a single-level population with a covariate
my_population <- declare_population(
  N = 6,
  female = rbinom(n = N, 1, prob = 0.5),
  height_ft = rnorm(N, mean = 5.67 - 0.33 * female, sd = 0.25)
)
```

```
# Declare a population from existing data

declare_population(mtcars)

# Resample from existing data

declare_population(N = 100, data = mtcars, handler = resample_data)


# Declare a two-level hierarchical population
# containing cities within regions and a
# pollution variable defined at the city level

my_population <- declare_model(
  regions = add_level(N = 5),
  cities = add_level(N = 10, pollution = rnorm(N, mean = 5))
)

# 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_population_function <- function(N) {
  data.frame(u = rnorm(N))
}

my_population_custom <- declare_model(N = 10, handler = my_population_function)
```

`declare_potential_outcomes` *Declare potential outcomes*

Description

Deprecated. Please use the potential_outcomes function within a declare_model declaration.

Usage

```
declare_potential_outcomes(
  ...,
  handler = potential_outcomes_handler,
  label = NULL
)

potential_outcomes_internal.formula(
```

```
  formula,
  conditions = c(0, 1),
  assignment_variables = "Z",
  data,
  level = NULL,
  label = outcome_variable
)

potential_outcomes_internal.NULL(
  formula = stop("Not provided"),
  ...,
  data,
  level = 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
`formula`	a formula to calculate potential outcomes as functions of assignment variables.
`conditions`	see `expand_conditions`. Provide values (e.g. `conditions = 1:4`) for a single assignment variable. If multiple assignment variables, provide named list (e.g. `conditions = list(Z1 = 0:1, Z2 = 0:1)`). Defaults to 0:1 if no conditions provided.
`assignment_variables`	The name of the assignment variable. Generally not required as names are taken from `conditions`.
`data`	a data.frame
`level`	a character specifying a level of hierarchy for fabricate to calculate at

Details

A `declare_potential_outcomes` function is used to create outcomes that each unit would express in each possible treatment condition.

Value

a function that returns a data.frame

Examples

```
# Potential outcomes can be declared in two ways:
# by using a formula or as separate variables.


# Using a formula
```

```
declare_model(N = 100, U = rnorm(N)) +
  declare_potential_outcomes(Y ~ 0.5*Z + U)

# As separate variables
declare_model(N = 100, U = rnorm(N)) +
  declare_potential_outcomes(Y_Z_0 = U,
                             Y_Z_1 = U + 0.5)
# (notice the naming structure: outcome_assignment_condition: Y_Z_1)


# You can change the name of the outcome
declare_model(N = 100, U = rnorm(N)) +
  declare_potential_outcomes(Y2 ~ 0.5*Z + U)

# You can change the name of the assignment_variable
declare_model(N = 100, U = rnorm(N)) +
  declare_potential_outcomes(Y ~ 0.5*D + U, assignment_variable = "D")


# `conditions` defines the "range" of the potential outcomes function
declare_model(N = 100, age = sample(18:65, N, replace = TRUE)) +
  declare_potential_outcomes(formula = Y ~ .05 + .25 * Z + .01 * age * Z,
                             conditions = 1:4)

# Multiple assignment variables can be specified in `conditions`. For example,
# in a 2x2 factorial potential outcome:

declare_model(N = 100, age = sample(18:65, N, replace = TRUE)) +
  declare_potential_outcomes(formula = Y ~ .05 + .25 * Z1 + .01 * age * Z2,
                             conditions = list(Z1 = 0:1, Z2 = 0:1))
```

`declare_reveal` *Declare a reveal outcomes step*

Description

Potential outcomes declarations indicate what outcomes would obtain for different possible values of assignment variables. But realized outcomes need to be "revealed." `declare_reveal` generates these realized outcomes using information on potential outcomes (for instance generated via `declare_potential_outcomes`) and the relevant assignment variables (for example created by `declare_assignment`). Revelation steps are usefully included after declaration of all assignments of conditions required to determine the realized outcome. If a revelation is not declared, DeclareDesign will try to guess appropriate revelations. Explicit revelation is recommended however.

Usage

```
declare_reveal(..., handler = declare_reveal_handler, label = NULL)
```

```
declare_reveal_handler(
  data = NULL,
  outcome_variables = Y,
  assignment_variables = Z,
  attrition_variables = 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
`data`	A data.frame containing columns for assignment and potential outcomes.
`outcome_variables`	The outcome prefix(es) of the potential outcomes.
`assignment_variables`	Unquoted name(s) of the assignment variable(s).
`attrition_variables`	Unquoted name of the attrition variable.

Details

This function was previously called `declare_reveal`. You can still use either one.

`declare_reveal` declares how outcomes should be realized. A "revelation" uses the random assignment to pluck out the correct potential outcomes (Gerber and Green 2012, Chapter 2). Revelation requires that every named outcome variable is a function of every named assignment variable within a step. Thus if multiple outcome variables depend on different assignment variables, multiple revelations are needed.

Examples

```
design <-
  declare_model(
    N = 100,
    U = rnorm(N),
    Y_Z_0 = U,
    Y_Z_1 = U + rnorm(N, mean = 2, sd = 2)
  ) +
  declare_assignment(Z = complete_ra(N, m = 50)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z))

# Declaring multiple assignment variables or multiple outcome variables

design   <-
  declare_model(
    N = 10,
    potential_outcomes(Y1 ~ Z),
```

```
    potential_outcomes(Y2 ~ 1 + 2 * Z),
    potential_outcomes(Y3 ~ 1 - X * Z, conditions = list(X = 0:1, Z = 0:1))
  ) +
  declare_assignment(Z = complete_ra(N)) +
  declare_assignment(X = complete_ra(N)) +
  declare_measurement(Y1 = reveal_outcomes(Y1 ~ Z),
                      Y2 = reveal_outcomes(Y2 ~ Z),
                      Y3 = reveal_outcomes(Y3 ~ X + Z))

design <-
  declare_model(
    N = 100,
    age = sample(18:95, N, replace = TRUE),
    potential_outcomes(Y ~ .25 * Z + .01 * age * Z),
    potential_outcomes(R ~ rbinom(n = N, size = 1, prob = pnorm(Y_Z_0)))
  ) +
  declare_assignment(Z = complete_ra(N, m = 25))
  declare_measurement(R = reveal_outcomes(R ~ Z),
                      Y = reveal_outcomes(Y ~ Z),
                      Y = ifelse(R == 1, Y, NA))
```

`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

```
design <- declare_model(

  classrooms = add_level(10),
  individuals = add_level(20, female = rbinom(N, 1, 0.5))

) + NULL

# Complete random sampling

design + declare_sampling(
  S = complete_rs(N = N, n = 50),
  filter = S == 1)

# equivalently, by default filter is set to S == 1
design + declare_sampling(S = complete_rs(N = N, n = 50),
                          legacy = FALSE)

# Stratified random sampling

design + declare_sampling(S = strata_rs(strata = female),
                          legacy = FALSE)
```

---

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 `model_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(model_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 model_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, model = lm_robust, model_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

```
design <-
  declare_model(
    N = 500,
    U = rnorm(N),
    Y_Z_0 = U,
    Y_Z_1 = U + rnorm(N, mean = 2, sd = 2)
  ) +
  declare_assignment(Z = complete_ra(N)) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_estimator(Y ~ Z, inquiry = my_inquiry) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z))

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

## End(Not run)

# You can choose your own diagnosands instead of the defaults e.g.,

my_diagnosands <-
  declare_diagnosands(median_bias = median(estimate - inquiry))
## Not run:
```

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

## End(Not run)
## Not run:
design <- set_diagnosands(design, diagnosands = my_diagnosands)
diagnosis <- diagnose_design(design)
diagnosis

## End(Not run)

# 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)
  )

# 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 = estimate > 0 & p.value <= alpha,
    mean_ci_length = mean(conf.high - conf.low)
  )
```

`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", "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

```
design <-
  declare_model(
    N = 500,
    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)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

## Not run:
# using built-in defaults:
diagnosis <- diagnose_design(design)
reshape_diagnosis(diagnosis, select = "Power")

## End(Not run)

## Not run:
# Adding a group for within group diagnosis:
diagnosis <- diagnose_design(design,
  make_groups = vars(significant = p.value <= 0.05),
  )
diagnosis

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

## End(Not run)

# using a user-defined diagnosand
```

```
my_diagnosand <- declare_diagnosands(absolute_error = mean(abs(estimate - estimand)))

## Not run:
diagnosis <- diagnose_design(design, diagnosands = my_diagnosand)
diagnosis

get_diagnosands(diagnosis)

get_simulations(diagnosis)


## End(Not run)
# Using an existing data frame of simulations
## Not run:
simulations <- simulate_design(designs, sims = 2)
diagnosis   <- diagnose_design(simulations_df = simulations_df)


## 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)
  )

# 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 = estimate > 0 & p.value <= alpha,
```

```
  mean_ci_length = mean(conf.high - conf.low)
)
```

---

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

```
design <-
  declare_model(
    N = 500,
    U = rnorm(N),
    Y_Z_0 = U,
    Y_Z_1 = U + rnorm(N, mean = 2, sd = 2)
  ) +
  declare_assignment(Z = complete_ra(N)) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

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

## End(Not run)

# using a user-defined diagnosand
my_diagnosand <- declare_diagnosands(
  absolute_error = mean(abs(estimate - estimand)))

## Not run:
diagnosis <- diagnose_design(design, diagnosands = my_diagnosand)
diagnosis
```

```
get_diagnosands(diagnosis)

get_simulations(diagnosis)

reshape_diagnosis(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

```
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)

dat_no_sampling <- draw_data(design, end = 3)

draw_estimands(design)

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)))

# with a custom designer function

designer <- function(N) {
  pop <-
    declare_model(
      N = N,
      U = rnorm(N),
      potential_outcomes(Y ~ 0.20 * Z + U)
    )
  assgn <- declare_assignment(Z = complete_ra(N, m = N/2))
  inquiry <- declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0))
  estimator <- declare_estimator(Y ~ Z, inquiry = inquiry)
  pop + pos + assgn + inquiry + estimator
}

# 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

design

## Not run:
insert_step(design, declare_step(dplyr::mutate, income = noise^2),
            after = my_assignment)
insert_step(design, declare_step(dplyr::mutate, income = noise^2),
            before = my_assignment)


# 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.

# get the labels for the steps
names(design)

insert_step(design,
  declare_sampling(S = complete_rs(N, n = 50),
  legacy = FALSE),
  after = "my_pop")

## End(Not run)


delete_step(design, my_assignment)
replace_step(design, my_assignment, declare_step(dplyr::mutate, words = "income"))
```

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 = TRUE, ...)

## 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

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

design

df <- draw_data(design)

estimates <- draw_estimates(design)
inquiries <- draw_estimands(design)

print_code(design)


my_population <- declare_model(N = 100)

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

my_design <- my_population + my_assignment

print_code(my_design)


my_model <-
  declare_model(
    N = 500,
    noise = rnorm(N),
    Y_Z_0 = noise,
    Y_Z_1 = noise + rnorm(N, mean = 2, sd = 2)
  )

my_sampling <- declare_sampling(S = complete_rs(N, n = 250))
```

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

my_inquiry <- declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0))

my_estimator <- declare_estimator(Y ~ Z, inquiry = my_inquiry)

my_reveal <- declare_measurement(Y = reveal_outcomes(Y ~ Z))

design <- my_model +
  my_sampling +
  my_inquiry +
  my_assignment +
  my_reveal +
  my_estimator

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

```
n <- 500
population <- declare_model(N = 1000)
sampling <- declare_sampling(S = complete_rs(N, n = n),
                             legacy = FALSE)
design <- population + sampling

# returns a single, modified design
modified_design <- redesign(design, n = 200)

# returns a list of six modified designs
design_vary_N <- redesign(design, n = seq(400, 900, 100))

# 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)

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

```
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)

reshape_diagnosis(diagnosis)

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

run_design *Run a design one time*

Description

Run a design one time

Usage

```
run_design(design)
```

Arguments

design	a DeclareDesign object

Examples

```
design <-
  declare_model(
    N = 100, X = rnorm(N),
    potential_outcomes(Y ~ (.25 + X) * Z + rnorm(N))
  ) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = complete_ra(N, m = 50)) +
  declare_measurement(Y = reveal_outcomes(Y ~ Z)) +
  declare_estimator(Y ~ Z, inquiry = "ATE")

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

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

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

```
design <-
declare_model(data = sleep) +
  declare_inquiry(mean_outcome = mean(extra)) +
  declare_sampling(S = complete_rs(N, n = 10)) +
  declare_estimator(extra ~ 1, inquiry = "mean_outcome",
     term = '(Intercept)', model = lm_robust)

diagnosands <- declare_diagnosands(
  median_bias = median(estimate - inquiry))

design <- set_diagnosands(design, diagnosands)

## Not run:
diagnose_design(design)

simulations_df <- simulate_design(design)

simulations_df <- set_diagnosands(simulations_df, design)

diagnose_design(simulations_df)


## 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

```
my_model <-
  declare_model(
    N = 500,
    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))

my_inquiry <- declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0))

my_estimator <- declare_estimator(Y ~ Z, inquiry = my_inquiry)

my_reveal <- declare_measurement(Y = reveal_outcomes(Y ~ Z))

design <- my_model +
  my_inquiry +
  my_assignment +
  my_reveal +
  my_estimator

## Not run:
simulations <- simulate_design(designs, sims = 2)
diagnosis <- diagnose_design(simulations_df = simulations)

## End(Not run)

## Not run:
# A fixed population with simulations over assignment only
head(simulate_design(design, sims = c(1, 1, 1, 100, 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