Package 'DeclareDesign'

January 7, 2019

Title Declare and Diagnose Research Designs

Version 0.14.0

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 (>= 3.4.0), randomizr (>= 0.16.1), fabricatr (>= 0.5.0), estimatr (>= 0.10.0)

Imports rlang, generics

License MIT + file LICENSE

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

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

Encoding UTF-8

LazyData true

RoxygenNote 6.1.1

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

NeedsCompilation no

Author Graeme Blair [aut, cre], Jasper Cooper [aut], Alexander Coppock [aut], Macartan Humphreys [aut], Neal Fultz [aut]

Maintainer Graeme Blair <graeme.blair@ucla.edu>

Repository CRAN

Date/Publication 2019-01-07 16:50:03 UTC

R topics documented:

`+.dd` *Add steps to create a design*

Description

Add steps to create a design

Usage

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

Arguments

lhs	A step in a research design, beginning with a function that draws the population. 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. Typically, many steps are declared using the declare_ functions, i.e., declare_population, declare_population, declare_sampling, declare_potential_outcomes, declare_estimand, declare_assignment, and declare_estimator.
rhs	A second step in a research design

Details

Users can supply three kinds of functions to create a design:

1. Data generating functions. These include population, assignment, and sampling functions.

2. Estimand functions.

3. Estimator functions.

The location of the estimand and estimator functions in the pipeline of functions determine *when* the values of the estimand and estimator are calculated. This allows users to, for example, differentiate between a population average treatment effect and a sample average treatment effect by placing the estimand function before or after sampling.

Design objects declared with the + operator can be investigated with a series of post-declaration commands, such as draw_data, draw_estimands, draw_estimates, and diagnose_design.

The print and summary methods for a design object return some helpful descriptions of the steps in your research design. If randomizr functions are used for any assignment or sampling steps, additional details about those steps are provided.

Value

a list of two functions, the design_function and the data_function. The design_function runs the design once, i.e. draws the data and calculates any estimates and estimands defined in ..., returned separately as two data.frame's. The data_function runs the design once also, but only returns the final data.

Examples

```
my_population <- declare_population(N = 500, noise = rnorm(N))

my_potential_outcomes <- declare_potential_outcomes(Y ~ Z + noise)

my_sampling <- declare_sampling(n = 250)

my_assignment <- declare_assignment(m = 25)

my_estimand <- declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0))

my_estimator <- declare_estimator(Y ~ Z, estimand = my_estimand)
```

```
my_mutate <- declare_step(dplyr::mutate, noise_sq = noise^2)

my_reveal <- declare_reveal()

design <- my_population + my_potential_outcomes + my_sampling +
          my_estimand + my_mutate +
          my_assignment + my_reveal + my_estimator

design

df <- draw_data(design)

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

# You can add steps to a design

design <- my_population + my_potential_outcomes
design + my_sampling

# Special Cases

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

design <- my_population + NULL
design


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

summary(diagnosis)

## End(Not run)
```

`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

---

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_population Population step

declare_potential_outcomes Potential outcomes step

declare_sampling Sampling step

declare_assignment Assignment step

declare_reveal Reveal outcomes step

declare_estimand Estimand step

declare_estimator Estimator step

Design Objects

+ Add steps to create a design

draw_data Simulate the DGP

run_design Simulate the DGP with estimands/estimators

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 assignment procedure*

Description

Declare assignment procedure

Usage

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

assignment_handler(data, ..., assignment_variable = "Z",
  append_probabilities_matrix = 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.
`assignment_variable`	Name for assignment variable (quoted). Defaults to "Z". Argument to be used with default handler.
`append_probabilities_matrix`	Should the condition probabilities matrix be appended to the data? Defaults to FALSE. Argument to be used with default handler.

Details

`declare_assignment` can work with any assignment_function that takes data and returns data. The default handler is `conduct_ra` from the `randomizr` package. This allows quick declaration of many assignment schemes that involve simple or complete random assignment with blocks and clusters. The arguments to `conduct_ra` can include `N`, `block_var`, `clust_var`, `m`, `m_each`, `prob`, `prob_each`, `block_m`, `block_m_each`, `block_prob`, `block_prob_each`, `num_arms`, and `conditions`. The arguments you need to specify are different for different designs. For details see the help files for `complete_ra`, `block_ra`, `cluster_ra`, or `block_and_cluster_ra`.

By default, `declare_assignment` declares a simple random assignment with probability 0.5.

Custom assignment handlers should augment the data frame with an appropriate column for the assignment(s).

Value

A function that takes a data.frame as an argument and returns a data.frame with additional columns appended including an assignment variable and (optionally) probabilities of assignment.

Examples

```
# Default handler delegates to conduct_ra

# Declare assignment of m units to treatment
my_assignment <- declare_assignment(m = 50)

# Declare assignment specifying assignment probability for each block
my_assignment <- declare_assignment(block_prob = c(1/3, 2/3), blocks = female)

# Declare assignment of clusters with probability 1/4
my_assignment <- declare_assignment(
  prob = 1/4,
  clusters = classrooms,
  assignment_variable = "X1"
)

# Declare factorial assignment (Approach 1):
#   Use complete random assignment to assign T1 and then use T1 as a block to assign T2.
 design <- declare_population(N = 4) +
   declare_assignment(assignment_variable = "T1") +
   declare_assignment(blocks = T1, assignment_variable = "T2")
   draw_data(design)

# Declare factorial assignment (Approach 2):
#   Assign to four conditions and then split into separate factors.
 design <- declare_population(N = 4) +
   declare_assignment(conditions = 1:4) +
   declare_step(fabricate, T1 = as.numeric(Z %in% 2:3), T2 = as.numeric(Z %in% 3:4))
   draw_data(design)

# Declare assignment using custom handler

custom_assignment <- function(data, assignment_variable = "X") {
 data[, assignment_variable] <- rbinom(n = nrow(data),
                                       size = 1,
                                       prob = 0.5)
 data
 }

 declare_population(N = 6) +
   declare_assignment(handler = custom_assignment, assignment_variable = "X")
```

`declare_estimand` *Declare estimand*

Description

Declares estimands. Estimands are the subjects of inquiry and can be estimated by an estimator.

Usage

```
declare_estimand(..., handler = estimand_handler, label = "estimand")

declare_estimands(..., handler = estimand_handler, label = "estimand")

estimand_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 `estimand_label` and `estimand` columns.

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

Value

a function that accepts a data.frame as an argument and returns a data.frame containing the value of the estimand.

Examples

```
# Set up a design stub for use in examples:

my_population  <- declare_population(N = 100, X = rnorm(N))
my_potential_outcomes  <- declare_potential_outcomes(
  Y ~ (.25 + X) * Z + rnorm(N))
my_assignment  <- declare_assignment(m = 50)
design_stub <- my_population + my_potential_outcomes + my_assignment +
  declare_reveal()

# Get example data to compute estimands on
dat <- draw_data(design_stub)

# ----------
# 1. Single estimand
# ----------

# Declare an average treatment effect (ATE) estimand
```

```
my_estimand_ATE <- declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0))
my_estimand_ATE(dat)

# or a conditional estimand

my_estimand_ATT <- declare_estimand(ATT = mean(Y_Z_1 - Y_Z_0),
                                    subset = (Z == 1))
my_estimand_ATT (dat)

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

my_estimator <- declare_estimator(Y ~ Z,
  estimand = my_estimand_ATE, label = "estimator")

design_one <- design_stub + my_estimand_ATE + my_estimator

draw_estimands(design_one)

# ----------
# 2. Multiple estimands
# ----------

# You can also specify multiple estimands for a single estimator

# With multiple estimands, you can use one estimator for both...

my_estimator_two <- declare_estimator(Y ~ Z,
  estimand = c(my_estimand_ATE, my_estimand_ATT))

design_two <- design_stub + my_estimand_ATE +
  my_estimand_ATT + my_estimator_two

draw_estimands(design_two)

# ----------
# 3. Paired estimands / estimators from a single model
# ----------

# For convenience you can also declare multiple estimands
# simultaneously and connect these to the corresponding
# terms for estimates used in the mode.

# Name your estimands the term name they get in your
# estimator, and set `term = TRUE`

estimands_regression <- declare_estimand(
  `(Intercept)` = mean(Y_Z_0),
  `Z` = mean(Y_Z_1 - Y_Z_0),
  term = TRUE,
  label="Regression_Estimands"
)

# For the model based estimator, specify the estimand as usual,
```

```
# but also set `term = TRUE`
estimators_regression <- declare_estimator(
  Y ~ Z,
  estimand = estimands_regression,
  model = lm,
  term = TRUE
)

design_regression <- design_stub + estimands_regression +
  estimators_regression

run_design(design_regression)

# ----------
# 4. Custom estimand function
# ----------

# You can declare more complex estimands by defining custom
# estimand functions:

estimand_function <- function(data, label) {
  ret <- with(data, median(Y_Z_1 - Y_Z_0))
  data.frame(estimand_label = label,
             estimand = ret,
             stringsAsFactors = FALSE)
}

estimand_custom <- declare_estimand(handler = estimand_function,
  label = "medianTE")

estimand_custom(dat)

# Use with custom estimators
estimator_function <- function(data){
  data.frame(estimate = with(data, median(Y)))
}
estimator_custom <-
  declare_estimator(handler = tidy_estimator(estimator_function),
                    estimand = estimand_custom)

design_custom <- design_stub + estimand_custom +
  estimator_custom

run_design(design_custom)

# ----------
# 5. Batch estimands and estimators
# ----------

# You can declare a group of estimands with distinct labels
# in one go and link them manually to a group of estimators.
# In this case you can add a \code{term} argument to the
# custom estimators to identify them.
```

```
f1 <- function(data) {
  data.frame(estimand_label = c("control", "ate"),
             estimand = with(data, c(mean(Y_Z_0), mean(Y_Z_1 - Y_Z_0))),
             stringsAsFactors = FALSE)
}
estimands <- declare_estimand(handler = f1)

f2 <- function(data) data.frame(estimate = with(data,
                              c(mean(Y[Z == 0]),
                              mean(Y[Z == 1]) - mean(Y[Z == 0]))),
                              term = 1:2)

estimators <- declare_estimator(handler = tidy_estimator(f2),
                              estimand = c("control", "ate"), label = "custom")

design      <- design_stub + estimands + estimators

## Not run:
diagnose_design(design, sims = 20, bootstrap_sims = FALSE,
                diagnosands = declare_diagnosands(
                select = c(mean_estimate, mean_estimand)))

## End(Not run)
```

`declare_estimator` *Declare estimator*

Description

Declares an estimator which generates estimates and associated statistics.

Usage

```
declare_estimator(..., handler = estimator_handler,
  label = "estimator")

declare_estimators(..., handler = estimator_handler,
  label = "estimator")

tidy_estimator(estimator_function)

model_handler(data, ..., model = estimatr::difference_in_means,
  term = FALSE)

estimator_handler(data, ..., model = estimatr::difference_in_means,
  term = FALSE, estimand = NULL, 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
`estimator_function`	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 label.
`data`	a data.frame
`model`	A model function, e.g. lm or glm. By default, the model is the `difference_in_means` function from the estimatr package.
`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 `!!`.
`estimand`	a declare_estimand step object, or a character label, or a list of either

Details

`tidy_estimator` takes an untidy estimation function, and returns a tidy handler which accepts standard labeling options.

The intent here is to factor out the estimator/estimand labeling so that it can be reused by other model handlers.

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.

Custom Estimators

`estimator_functions` implementations should be tidy (accept and return a data.frame)

`model` implementations should at a minimum provide S3 methods for `summary` and `confint`.

Examples

```
# Declare estimand
my_estimand <- declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0))

# Declare estimator using the default handler using `difference_in_means`
# estimator from `estimatr` package. Returns the first non-intercept term
# as estimate

my_estimator_dim <- declare_estimator(Y ~ Z, estimand = "ATE", label = "DIM")

# Use lm function from base R
my_estimator_lm <- declare_estimator(Y ~ Z, estimand = "ATE",
```

```
  model = lm, label = "LM")
# Use lm_robust (linear regression with robust standard errors) from
# `estimatr` package

my_estimator_lm_rob <- declare_estimator(
  Y ~ Z,
  estimand = "ATE",
  model = lm_robust,
  label = "LM_Robust"
)

# Set `term` if estimate of interest is not the first non-intercept variable
my_estimator_lm_rob_x <- declare_estimator(
  Y ~ X + Z,
  estimand = my_estimand,
  term = "Z",
  model = lm_robust
)

# Use glm from base R
my_estimator_glm <- declare_estimator(
  Y ~ X + Z,
  family = "gaussian",
  estimand = my_estimand,
  term = "Z",
  model = glm
)

# A probit
estimator_probit <- declare_estimator(
  Y ~ Z,
  model = glm,
  family = binomial(link = "probit"),
  term = "Z"
)

# Declare estimator using a custom handler

# Define your own estimator and use the `tidy_estimator` function for labeling
# Must have `data` argument that is a data.frame
my_estimator_function <- function(data){
  data.frame(estimate = with(data, mean(Y)))
}

my_estimator_custom <- declare_estimator(
  handler = tidy_estimator(my_estimator_function),
  estimand = my_estimand
)

# Customize labeling

my_estimator_function <- function(data){
  data.frame(
```

```
    estimator_label = "foo",
    estimand_label = "bar",
    estimate = with(data, mean(Y)),
    n = nrow(data),
    stringsAsFactors = FALSE
  )
}

my_estimator_custom2 <- declare_estimator(handler = my_estimator_function)


# Examples

# First, set up the rest of a design
set.seed(42)

design_def <-
  declare_population(N = 100, X = rnorm(N), W = rexp(N, 1), noise = rnorm(N)) +
  declare_potential_outcomes(Y ~ .25 * Z + noise) +
  declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(m = 50) +
  declare_reveal() +
  my_estimator_dim

draw_estimates(design_def)

# Can also use declared estimator on a data.frame
dat <- draw_data(design_def)
my_estimator_dim(dat)

# ----------
# 2. Using existing estimators
# ----------

design <- replace_step(design_def, my_estimator_dim, my_estimator_lm_rob)
draw_estimates(design)

design <- replace_step(design_def, my_estimator_dim, my_estimator_lm)
draw_estimates(design)

design <- replace_step(design_def, my_estimator_dim, my_estimator_glm)
draw_estimates(design)

# ----------
# 3. Using custom estimators
# ----------

design <- replace_step(design_def, my_estimator_dim, my_estimator_custom)

draw_estimates(design)

# The names in your custom estimator return should match with
# your diagnosands when diagnosing a design
```

```
my_median <- function(data) data.frame(med = median(data$Y))

my_estimator_median <- declare_estimator(
  handler = tidy_estimator(my_median),
  estimand = my_estimand
)

design <- replace_step(design_def, my_estimator_dim, my_estimator_median)

draw_estimates(design)

my_diagnosand <- declare_diagnosands(med_to_estimand = mean(med - estimand),
  keep_defaults = FALSE)

## Not run:
diagnose_design(design, diagnosands = my_diagnosand, sims = 5,
  bootstrap_sims = FALSE)

## End(Not run)

# ----------
# 4. Multiple estimators per estimand
# ----------

design_two <- insert_step(design_def,  my_estimator_lm,
  after = my_estimator_dim)

draw_estimates(design_two)

## Not run:
diagnose_design(design_two, sims = 5, bootstrap_sims = FALSE)

## End(Not run)
```

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

Description

Declare the size and features of the population

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 function that returns a data.frame.

Examples

```
# Default handler is fabricatr::fabricate

# 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,
  gender = draw_binary(N, prob = 0.5),
  height_ft = rnorm(N, mean = 5 + 4/12 + 4/12 * gender, sd = 3/12)
)
my_population()

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

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

# Declare a population using a custom function

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

my_population_custom <- declare_population(N = 10, handler = my_population_function)

my_population_custom()
```

---

declare_potential_outcomes

Declare potential outcomes

Description

Declare potential outcomes

Usage

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

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

potential_outcomes.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` declaration returns a function. The function takes and returns a data.frame with potential outcomes columns appended. These columns describe the outcomes that each unit would express if that unit were in the corresponding treatment condition.

Declaring a potential outcomes function requires postulating a particular causal process. One can then assess how designs fare under the postulated process. Multiple processes can be considered in a single design or across design. For instance one could declare two processes that rival theories would predict.

Potential outcomes can be declared as separate variables or by using a formula. See examples below.

Value

a function that returns a data.frame

Examples

```
# Declare potential outcomes using default handler

# There are two ways of declaring potential outcomes:

# As separate variables

my_potential_outcomes <- declare_potential_outcomes(
  Y_Z_0 = .05,
  Y_Z_1 = .30 + .01 * age
)

# Using a formula
 my_potential_outcomes <- declare_potential_outcomes(
   Y ~ .05 + .25 * Z + .01 * age * Z)

# `conditions` defines the "range" of the potential outcomes function
 my_potential_outcomes <- 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:

 my_potential_outcomes <- declare_potential_outcomes(
   formula = Y ~ .05 + .25 * Z1 + .01 * age * Z2,
   conditions = list(Z1 = 0:1, Z2 = 0:1)
 )

# You can also declare potential outcomes using a custom handler

my_po_function <- function(data) {
  data$Y_treated   <- rexp(nrow(data), .2)
  data$Y_untreated <- rexp(nrow(data), .4)
  data
}

custom_potential <- declare_potential_outcomes(handler = my_po_function)
```

---

declare_reveal *Declare a reveal outcomes step*

Description

Potential outcomes declarations indicate what outcomes would obtain for different possible values of assignment variables. To reveal actual outcomes we combine assignments with potential outcomes. `declare_reveal` provides information on how this revelation should be implemented,

identifying the relevant assignment variables (for example created by `declare_assignment`) and outcome variables. 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 though explicit revelation is recommended.

Usage

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

reveal_outcomes_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

`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). If you create a simple design (with assignment variable Z and outcome variable Y) with the + operator but omit a reveal declaration, DeclareDesign will attempt to insert a revelation step automatically. If you have multiple outcomes to reveal or different names for the outcome or assignment variables, use `declare_reveal` to customize which outcomes are revealed. 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

```
my_population <- declare_population(N = 100, noise = rnorm(N))

my_potential_outcomes <- declare_potential_outcomes(
  Y_Z_0 = noise, Y_Z_1 = noise +
  rnorm(N, mean = 2, sd = 2))

my_assignment <- declare_assignment(m = 50)

my_reveal <- declare_reveal()
```

```
design <- my_population +
  my_potential_outcomes +
  my_assignment +
  my_reveal

design

#  Here the + operator results in the same design being
#  created, because it automatically adds a declare_reveal step.

design <- my_population + my_potential_outcomes + my_assignment

# Declaring multiple assignment variables or multiple outcome variables

population   <- declare_population(N = 10)
potentials_1 <- declare_potential_outcomes(Y1 ~ Z)
potentials_2 <- declare_potential_outcomes(Y2 ~ 1 + 2*Z)
potentials_3 <- declare_potential_outcomes(Y3 ~ 1 - X*Z, conditions = list(X = 0:1, Z = 0:1))
assignment_Z <- declare_assignment(assignment_variable = "Z")
assignment_X <- declare_assignment(assignment_variable = "X")
reveal_1    <- declare_reveal(outcome_variables = c("Y1", "Y2"), assignment_variables = "Z")
reveal_2    <- declare_reveal(outcome_variables = "Y3", assignment_variables = c("X", "Z"))

# Note here that the reveal cannot be done in one step, e.g. by using
# declare_reveal(outcome_variables = c("Y1", "Y2", "Y3"),
#   assignment_variables = c("X","Z"))
# The reason is that in each revelation all outcome variables should be a
# function of all assignment variables.

# declare_reveal can also be used to declare outcomes that include attrition

population <- declare_population(N = 100, age = sample(18:95, N, replace = TRUE))

potential_outcomes_Y <- declare_potential_outcomes(Y ~ .25 * Z + .01 * age * Z)

assignment <- declare_assignment(m = 25)

potential_outcomes_attrition <-
  declare_potential_outcomes(R ~ rbinom(n = N, size = 1, prob = pnorm(Y_Z_0)))

reveal_attrition <- declare_reveal(outcome_variables = "R")
reveal_outcomes <- declare_reveal(outcome_variables = "Y", attrition_variables = "R")

my_design <- population + potential_outcomes_Y + potential_outcomes_attrition +
  my_assignment + reveal_attrition + reveal_outcomes
```

---

declare_sampling *Declare sampling procedure*

Description

Declare sampling procedure

Usage

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

sampling_handler(data, ..., sampling_variable = "S")
```

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.
sampling_variable	The prefix for the sampling inclusion probability variable.

Details

declare_sampling can work with any sampling_function that takes data and returns data. The default handler is draw_rs from the randomizr package. This allows quick declaration of many sampling schemes that involve strata and clusters.

The arguments to draw_rs can include N, strata_var, clust_var, n, prob, strata_n, and strata_prob. The arguments you need to specify are different for different designs.

Note that declare_sampling works similarly to declare_assignment a key difference being that declare_sampling functions subset data to sampled units rather than simply appending an indicator for membership of a sample (assignment). If you need to sample but keep the dataset use declare_assignment and define further steps (such as estimation) with respect to subsets defined by the assignment.

For details see the help files for complete_rs, strata_rs, cluster_rs, or strata_and_cluster_rs

Value

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

```
# Default handler is `draw_rs` from `randomizr` package

# Simple random sampling
my_sampling <- declare_sampling(n = 50)

# Stratified random sampling
my_stratified_sampling <- declare_sampling(strata = female)
```

```
# Custom random sampling functions

my_sampling_function <- function(data, n=nrow(data)) {
   data[sample(n,n,replace=TRUE), , drop=FALSE]
}

my_sampling_custom <- declare_sampling(handler = my_sampling_function)

my_sampling_custom(sleep)
```

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

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

deprecated *Deprecated functions*

Description

The function `get_estimands` has been replaced with `draw_estimands`.

Usage

```
get_estimands(...)
```

Arguments

`...`	options sent to the old version of `get_estimands`.

diagnosand_handler *Declare diagnosands*

Description

Declare diagnosands

Usage

```
diagnosand_handler(data, ..., select, subtract, keep_defaults = TRUE,
  subset = NULL, alpha = 0.05, label)

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

Arguments

`data`	A data.frame.
`...`	A set of new diagnosands.
`select`	A set of the default diagnosands to report e.g., `select = c(bias, rmse)`.
`subtract`	A set of the default diagnosands to exclude e.g., `subtract = c(bias, rmse)`. Do not provide values for both `select` and `subtract`.
`keep_defaults`	A flag for whether to report the default diagnosands. Defaults to `TRUE`.
`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 `estimand_label` 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 estimand. Many diagnosands will be a function of these variables.

By default (`keep_defaults = TRUE`), a set of common diagnosands are reported:

bias = mean(estimate - estimand)
rmse = sqrt(mean((estimate - estimand)^2))
power = mean(p.value < .05)
coverage = mean(estimand <= conf.high & estimand >= conf.low)
mean_estimate = mean(estimate)
sd_estimate = sd(estimate)
type_s_rate = mean((sign(estimate) != sign(estimand))[p.value < alpha])
mean_estimand = mean(estimand)

Value

a function that returns a data.frame

Examples

```
my_population <- declare_population(N = 500, noise = rnorm(N))

my_potential_outcomes <- declare_potential_outcomes(
  Y_Z_0 = noise, Y_Z_1 = noise +
  rnorm(N, mean = 2, sd = 2))

my_assignment <- declare_assignment()

my_estimand <- declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0))

my_estimator <- declare_estimator(Y ~ Z, estimand = my_estimand)

my_reveal <- declare_reveal()

design <- my_population + my_potential_outcomes + my_estimand +
        my_assignment + my_reveal + my_estimator

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

## End(Not run)

# You can select a set of those diagnosands via the \code{select} argument e.g.,
```

```
my_diagnosands <- declare_diagnosands(select = c(bias, rmse))

# Alternatively, you can report all of the default diagnosands and subtract a subset of them e.g.,

my_diagnosands <- declare_diagnosands(subtract = type_s_rate)

# You can add your own diagnosands in addition to or instead of the defaults e.g.,

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

# or to report only \code{median_bias}

my_diagnosands <-
   declare_diagnosands(median_bias = median(estimate - estimand),
                       keep_defaults = FALSE)

# Below is the code that makes the default diagnosands.
# You can use these as a model when writing your own diagnosands.

default_diagnosands <- declare_diagnosands(
bias = mean(estimate - estimand),
rmse = sqrt(mean((estimate - estimand) ^ 2)),
power = mean(p.value < alpha),
coverage = mean(estimand <= conf.high & estimand >= conf.low),
mean_estimate = mean(estimate),
sd_estimate = sd(estimate),
mean_se = mean(std.error),
type_s_rate = mean((sign(estimate) != sign(estimand))[p.value < alpha]),
mean_estimand = mean(estimand)
)
```

`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, add_grouping_variables = NULL)

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

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 estimand(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.
add_grouping_variables	Variables used to generate groups of simulations for diagnosis. Added to list default list: c("design_label", "estimand_label", "estimator_label", "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

```
my_population <- declare_population(N = 500, noise = rnorm(N))

my_potential_outcomes <- declare_potential_outcomes(
  Y_Z_0 = noise, Y_Z_1 = noise +
  rnorm(N, mean = 2, sd = 2))

my_assignment <- declare_assignment()

my_estimand <- declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0))

my_reveal <- declare_reveal()

my_estimator <- declare_estimator(Y ~ Z, estimand = my_estimand)

design <- my_population +
   my_potential_outcomes +
   my_estimand +
```

```
   my_assignment +
   my_reveal +
   my_estimator

## 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)


## 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)
```

`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

```
my_population <- declare_population(N = 500, noise = rnorm(N))

my_potential_outcomes <- declare_potential_outcomes(
  Y_Z_0 = noise, Y_Z_1 = noise +
  rnorm(N, mean = 2, sd = 2))

my_assignment <- declare_assignment()

my_estimand <- declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0))

my_estimator <- declare_estimator(Y ~ Z, estimand = my_estimand)

my_reveal <- declare_reveal()

design <- my_population +
  my_potential_outcomes +
  my_estimand +
  my_assignment +
  my_reveal +
  my_estimator

## 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)
```

`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_population(N = N, noise = rnorm(N))
  pos <- declare_potential_outcomes(Y ~ 0.20 * Z + noise)
  assgn <- declare_assignment(m = N / 2)
  mand <- declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0))
  mator <- declare_estimator(Y ~ Z, estimand = mand)
  pop + pos + assgn + mand + mator
}

# 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)
```

`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_population <- declare_population(N = 100, noise = rnorm(N), label = "my_pop")

my_potential_outcomes <-
  declare_potential_outcomes(Y_Z_0 = noise,
                             Y_Z_1 = noise + rnorm(N, mean = 2, sd = 2))

my_assignment <- declare_assignment(m = 50)
```

```
my_assignment_2 <- declare_assignment(m = 25)

design <- my_population + my_potential_outcomes + my_assignment

design

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(n = 50), after = "my_pop")


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

---

post_design *Explore your design*

Description

Explore your design

Usage

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

draw_data(design)

draw_estimands(...)

draw_estimates(...)

## 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
data	A data.frame object with sufficient information to run estimators.

start	(Defaults to 1) a scalar indicating which step in the design to begin getting estimates from. By default all estimators are calculated, 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 getting estimates from.
...	optional arguments to be sent to summary function
x	a design object, typically created using the + operator
verbose	an indicator for printing a long summary of the design, defaults to TRUE
object	a design object created using the + operator

Examples

```
design <-
  declare_population(N = 500, noise = rnorm(N)) +
  declare_potential_outcomes(Y ~ noise + Z * rnorm(N, 2, 2)) +
  declare_sampling(n = 250) +
  declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_step(dplyr::mutate, noise_sq = noise^2) +
  declare_assignment(m = 25) +
  declare_reveal() +
  declare_estimator(Y ~ Z, estimand = "my_estimand")

design

df <- draw_data(design)

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


my_population <- declare_population(N = 500, noise = rnorm(N))

my_potential_outcomes <- declare_potential_outcomes(
  Y_Z_0 = noise, Y_Z_1 = noise +
  rnorm(N, mean = 2, sd = 2))

my_sampling <- declare_sampling(n = 250)

my_assignment <- declare_assignment(m = 25)

my_estimand <- declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0))

my_estimator <- declare_estimator(Y ~ Z, estimand = my_estimand)

my_mutate <- declare_step(dplyr::mutate, noise_sq = noise ^ 2)

my_reveal <- declare_reveal()

design <- my_population +
  my_potential_outcomes +
```

```
  my_sampling +
  my_estimand +
  my_mutate +
  my_assignment +
  my_reveal +
  my_estimator

summary(design)
```

`print_code` *Print code to recreate a design*

Description

Print code to recreate a design

Usage

```
print_code(design)
```

Arguments

`design`	A design object, typically created using the + operator

Examples

```
my_population <- declare_population(N = 100)

my_assignment <- declare_assignment(m = 50)

my_design <- my_population + my_assignment

print_code(my_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_population(N = 1000)
sampling <- declare_sampling(n = n)
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(prob_each = prob_each)

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_population(N = 1, A = X)       + NULL
design2 <- declare_population(N = 1, A = f(2, X)) + NULL
design3 <- declare_population(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

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)
```

Arguments

`diagnosis`	An object from `diagnose_design`, either a diagnosand dataframe or a list containing a diagnosand dataframe
`digits`	Number of digits.
`select`	List of columns to include in output. Defaults to all.

Value

A formatted text table with bootstrapped standard errors in parentheses.

Examples

```
# library(DesignLibrary)
# diagnosis <- diagnose_design(two_arm_designer(), sims = 3)
# reshape_diagnosis(diagnosis)
# reshape_diagnosis(diagnosis, select = c("Bias", "Power"))
```

run_design *Execute a design*

Description

Execute a design

Usage

```
run_design(design)
```

Arguments

design	a DeclareDesign object

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_population(data = sleep) +
  declare_sampling(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(design, diagnosands = default_diagnosands)
```

Arguments

`design`	A design typically created using the + operator
`diagnosands`	A set of diagnosands created by `declare_diagnosands`

Value

a design object with a diagnosand attribute

Examples

```
design <-
declare_population(data = sleep) +
  declare_estimand(mean_outcome = mean(extra)) +
  declare_sampling(n = 10) +
  declare_estimator(extra ~ 1, estimand = "mean_outcome",
     term = '(Intercept)', model = lm_robust)
```

```
diagnosands <- declare_diagnosands(
  median_bias = median(estimate - estimand), keep_defaults = FALSE)

design <- set_diagnosands(design, diagnosands)

## Not run:
diagnose_design(design)

## 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_population <- declare_population(N = 500, noise = rnorm(N))

my_potential_outcomes <- declare_potential_outcomes(
  Y_Z_0 = noise, Y_Z_1 = noise +
  rnorm(N, mean = 2, sd = 2))
```

```
my_assignment <- declare_assignment()

my_estimand <- declare_estimand(ATE = mean(Y_Z_1 - Y_Z_0))

my_estimator <- declare_estimator(Y ~ Z, estimand = my_estimand)

my_reveal <- declare_reveal()

design <- my_population +
  my_potential_outcomes +
  my_estimand +
  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)
```

Index