

Package: statim (via r-universe)

September 21, 2026

Type Package

Title A Declarative Interface for Statistical Inference

Version 0.1.0.9000

Description A declarative interface for statistical inference built on the S7 object system. The layouts of the statistical inference are described using expressive variable mappers, null hypotheses are stated as mathematical expressions over population parameters, and estimation is deferred until explicitly requested. Supports estimation method involving hypothesis testing and model-based inference with an extensible variant system for registering custom estimation methods.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Imports rlang (>= 1.2.0), cli, vctrs (>= 0.7.0), tibble, S7, tidyselect, stats, utils, tabstats (>= 0.2.0), pillar

Suggests testthat (>= 3.0.0), magrittr, box, rmarkdown, broom, pander, Rfast2, purrr, tidyr, dplyr, forcats, ggplot2, patchwork, sysfonts, showtext, ggdist, ggtext, boot, knitr, fansi, lintr, rstatix, readr, withr

Collate 'statim-package.R' 'ops.R' 'utils.R' 'add-option.R' 'add-stat-define.R' 'zzz.R' 'define-var-ids.R' 'define-var-ids-process.R' 'define-var-ids-info.R' 'define-model.R' 'define-var-id-process-helpers.R' 'generate-pairs.R' 'inject-and-run.R' 'stat-construct.R' 'stat-infer-class.R' 'baseline-variant.R' 'stat-spec.R' 'htest-prepare.R' 'model-prepare.R' 'stat-prepare.R' 'recalibrator-via.R' 'hypothesis-params.R' 'hypothesis-parser.R' 'hypothesis-helpers.R' 'hypothesis-mappers.R' 'hypothesis-core.R' 'hypothesis-validators.R' 'stat-exec.R' 'tidy-core.R' 'predict-core.R' 'gauge-core.R' 'htest-impl-container.R' 'htest-proptest-defs.R' 'htest-proptest.R'

```
'htest-ttest-defs-on.R' 'htest-ttest-defs-xby.R'
'htest-ttest-defs-formula.R' 'htest-ttest-defs-pairwise.R'
'htest-ttest-one.R' 'htest-ttest.R' 'tidy-ttest-def-two.R'
'tidy-ttest-def-formula.R' 'gauge-ttest-def-formula.R'
'htest-testcor-defs-rel.R' 'htest-testcor-defs-formula.R'
'htest-testcor.R' 'model-impl-container.R' 'model-expand.R'
'model-anova-mod.R' 'model-linear-reg-defs.R'
'model-linear-reg.R' 'model-glm-defs.R' 'model-glm.R'
'display-meta.R' 'tidy-auto.R' 'predict-auto.R' 'gauge-auto.R'
'global_variables.R'
```

Config/testthat/edition 3

Roxygen list(markdown = TRUE)

Depends R (>= 4.1.0)

URL <https://s7-stats.github.io/statim/>,
<https://github.com/s7-stats/statim>

BugReports <https://github.com/s7-stats/statim/issues>

RoxygenNote 7.3.3

VignetteBuilder knitr

Repository <https://s7-stats.r-universe.dev>

Date/Publication 2026-08-07 23:11:48 UTC

RemoteUrl <https://github.com/s7-stats/statim>

RemoteRef HEAD

RemoteSha 0760a9b902520b1b6bdb91a6abfc7e4bd7407300

Contents

add-stat-define	4
add-variant	6
agendas	7
anova-mod	8
auto_gauge	9
auto_predict	10
auto_tidy	11
baseline	12
claim-vars-validators	13
claim_contrast_coefs	15
claim_scalar	16
class_corr_two	16
class_glm_object	17
class_lm_object	19
class_p_test	21
class_stat_infer	22
class_ttest_one	24

class_ttest_pairwise	25
class_ttest_two	26
class_var_inform	27
conclude	27
COR_TEST	29
display	30
gauge	31
GLM	32
HTEST_FN	33
inlines	34
layout-define-base	35
LINEAR_REG	36
making_gauge	37
making_predict	38
making_tidy	39
map_claim	39
method_gauge	40
method_predict	41
method_tidy	41
model-processor	42
MODEL_FN	43
modifying-assignment	43
MU	44
null-hyp	45
on	45
P_TEST	46
pairwise	48
param_obj	49
PI	50
predict	50
prepare	51
prepare-model	52
prepare-test	53
prop	54
purge_stat_defines	54
rel	55
RHO	55
stat-infer-definer	56
STAT_CONSTRUCTOR	57
T_TEST	57
tidy	59
var_id	60
var_id_info	61
variant	62
via	63
write_models	64
x_by	66

add-stat-define	<i>Add or remove stat_define implementations on a test or model function</i>
-----------------	--

Description

These are developer-interface functions intended for package authors extending the `statim` framework with new model types.

`add_stat_define()` registers a new `stat_define()` for a stat function, enabling it to handle a previously unsupported variable mapper `<var_id>`. Registering a model type that already exists — whether baked-in or previously registered — is an error.

`remove_stat_define()` removes a previously registered "user"-originated entry. "package"-scoped entries are self-cleaning via `purge_stat_defines()` called in the registering package's `.onUnload()`.

Usage

```
add_stat_define(
  stat_fn,
  model_type,
  impl,
  compatible_params = list(),
  origin = c("user", "package"),
  .pkg = NULL
)

remove_stat_define(stat_fn, model_type)
```

Arguments

<code>stat_fn</code>	A test or model function built with <code>HTEST_FN()</code> or <code>MODEL_FN()</code> (e.g. <code>T_TEST</code> , <code>P_TEST</code>).
<code>model_type</code>	An S7 <code><var_id></code> class (e.g. <code>x_by</code> , <code>S7::class_formula</code>).
<code>impl</code>	An <code>agendas()</code> object.
<code>compatible_params</code>	A list of param S7 classes (e.g. <code>list(MU)</code>). Defaults to <code>list()</code> .
<code>origin</code>	One of "user" (default) or "package". Use "user" for interactive or script-level registration scoped to the current session. Use "package" inside your package's <code>.onLoad()</code> , paired with <code>.pkg = pkgname</code> — see the Package authors section below.
<code>.pkg</code>	The registering package name. Required when <code>origin = "package"</code> ; ignored otherwise. Pass the <code>pkgname</code> argument that R supplies to <code>.onLoad()</code> / <code>.onAttach()</code> . This is used to attribute the entry and to enable <code>purge_stat_defines()</code> to clean it up on unload.

Value

NULL, invisibly.

Scoping and lifecycle

Registrations have two scopes:

- "user"-*scoped* entries live for the duration of the R session (or until explicitly removed with [remove_stat_define\(\)](#)). Use this for interactive work or in scripts.
- "package"-*scoped* entries are intended for package authors who ship extensions to statim. They must be registered in `.onLoad()` and cleaned up in `.onUnload()` via [purge_stat_defines\(\)](#). This keeps the registry tidy when the extending package is unloaded.

Package authors

To ship an extension as a package, register in `zzz.R` (or any file loaded early):

```
.onLoad = function(libname, pkgname) {
  statim::add_stat_define(
    P_TEST,
    my_var_id,
    impl = agendas(
      base = baseline(
        fn = function(.proc, .p = 0.5, .alt = "two.sided", .ci = 0.95) {
          # your implementation
        }
      )
    ),
    compatible_params = list(PI),
    origin = "package",
    .pkg = pkgname
  )
}

.onUnload = function(libpath) {
  statim::purge_stat_defines("yourpackage")
}
```

The `.pkg = pkgname` argument is how `statim` knows which package owns the entry. Without it, `origin = "package"` is an error. Never hard-code the string yourself — always forward the `pkgname` R passes to `.onLoad()`.

See Also

[stat_define\(\)](#), [agendas\(\)](#), [remove_stat_define\(\)](#), [purge_stat_defines\(\)](#)

Examples

```
# Session-scoped registration (interactive use or scripts)
mt = S7::new_class("my_var", parent = var_id)

add_stat_define(
  P_TEST,
  mt,
  impl = agendas(
    base = baseline(
      fn = function(.proc, .value = 1) list(value = .value)
    )
  )
)

# Clean up when done
remove_stat_define(P_TEST, mt)
```

add-variant

Add or remove variant implementations on a test or model function

Description

These are **developer-interface** functions intended for package authors extending the `statim` framework with new method variants.

`add_variant()` is used as the left-hand side of the `%<-%` operator to register a `variant()` for a stat function and model type. "default" is frozen and cannot be added.

`remove_variant()` removes a previously registered "user"-originated variant. "package"-level entries are self-cleaning: they exist for the duration the registering package is loaded and vanish when it is unloaded.

Usage

```
add_variant(obj, model_type, name, origin = c("user", "package"))
```

```
remove_variant(obj, model_type, name)
```

Arguments

<code>obj</code>	A test or model function built with <code>HTEST_FN()</code> or <code>MODEL_FN()</code> (e.g. <code>T_TEST</code>). Used to scope the registry key.
<code>model_type</code>	An <code>S7 <var_id></code> class (e.g. <code>x_by</code> , <code>S7::class_formula</code>).
<code>name</code>	A string naming the variant to add.
<code>origin</code>	One of "user" (default, session-scoped) or "package" (load-scoped, intended for <code>.onLoad()</code>).

Value

An `add_variant_call` object, consumed by `%<-%`.

See Also

`stat_define()`, `variant()`, `agendas()`, `model_processor()`

Examples

```
# Add a bootstrap variant for x_by (user level).
# .proc$x_data[[1]] is the response vector; .proc$group_data is the
# grouping data frame. See ?model_processor for all available keys.
add_variant(T_TEST, x_by, "another_boot") %<-% variant(
  fn = function(.proc, .n = 1000L) {
    x = .proc$x_data[[1]]
    grp = as.character(.proc$group_data[[1]])
    lvls = unique(grp)
    x1 = x[grp == lvls[[1]]]
    x2 = x[grp == lvls[[2]]]
    boot_fn = function(d, i) mean(d[i, 1]) - mean(d[i, 2])
    b = boot::boot(data.frame(x1, x2), boot_fn, R = .n)
    boot::boot.ci(b, type = "perc")
  }
)

# Remove it, returning to the original slate
remove_variant(T_TEST, x_by, "another_boot")

# Package level (inside .onLoad())
add_variant(T_TEST, x_by, "another_boot", origin = "package") %<-% variant(
  fn = function(.proc, .n = 1000L) { ... }
)
```

agendas

Collect implementations for a statistical procedure

Description

`agendas()` is the container for all implementations of a procedure. Requires exactly one `baseline()` and accepts any number of named `variant()` objects.

Usage

```
agendas(base, ...)
```

Arguments

<code>base</code>	A <code>baseline()</code> object. Required.
<code>...</code>	Named <code>variant()</code> objects.

Value

An agendas S3 object.

See Also

[baseline\(\)](#), [variant\(\)](#), [stat_define\(\)](#)

anova-mod	<i>ANOVA table for linear model comparisons</i>
-----------	---

Description

`anova()` computes an incremental F-test across two or more fitted linear models. It dispatches on four input types:

Usage

```
anova(object, ..., test = "F")
```

Arguments

<code>object</code>	A <code>multi_lazy</code> , <code>anova_lazy</code> , <code>model_lazy</code> , or <code>cld_exec</code> object.
<code>...</code>	Additional <code>model_lazy</code> or <code>cld_exec</code> objects.
<code>test</code>	A string. One of "F" (default), "LRT", or "Chisq".

Format

An object of class `S7_external_generic` of length 4.

Details

- A `multi_lazy` from `write_models()` |> `prepare_model()`.
- An `anova_lazy` from `write_models()` |> `prepare_model()` (legacy path, kept for backward compatibility).
- One or more `model_lazy` objects from `prepare_model()`.
- One or more `cld_exec` objects from `conclude()`.

Value

A `cld_anova` object, invisibly.

See Also

[write_models\(\)](#), [prepare_model\(\)](#), [conclude\(\)](#)

Examples

```
# via write_models()
LifeCycleSavings |>
  write_models(
    f1 = sr ~ 1,
    f2 = sr ~ pop15,
    f3 = sr ~ pop15 + pop75,
    f4 = sr ~ pop15 + pop75 + dpi,
    f5 = sr ~ pop15 + pop75 + dpi + ddpi
  ) |>
  prepare_model(LINEAR_REG) |>
  anova()

# via model_lazy
mod1 = LifeCycleSavings |> define_model(sr ~ 1) |> prepare_model(LINEAR_REG)
mod2 = LifeCycleSavings |> define_model(sr ~ pop15) |> prepare_model(LINEAR_REG)

anova(mod1, mod2)

# via conclude()
mod1 = LifeCycleSavings |> define_model(sr ~ 1) |> prepare_model(LINEAR_REG) |> conclude()
mod2 = LifeCycleSavings |> define_model(sr ~ pop15) |> prepare_model(LINEAR_REG) |> conclude()

anova(mod1, mod2)
anova(mod1, mod2, test = "LRT")
```

auto_gauge

Automatically gauge effect size from a statistical result

Description

`auto_gauge()` is the protocol generic for computing effect-size quantities from result objects produced by `fn` in `baseline()` and `variant()`. It is called automatically by `gauge()` when the result stored in `cld_exec@data` is a `class_stat_infer` subclass.

Usage

```
auto_gauge(x, ...)
```

Arguments

<code>x</code>	A <code>class_stat_infer</code> subclass object, typically <code>cld_exec@data</code> .
<code>...</code>	Currently unused. Passed to the dispatched method.

Details

Register a method on your output class to participate:

```
S7::method(auto_gauge, my_test_result) = function(x, ...) {
  tibble::tibble(metric = "cohens_d", value = ...)
}
```

A variant whose fn returns the same result class as baseline inherits auto_gauge() for free via S7's parent chain.

Value

A tibble with metric and value columns, one row per effect-size quantity.

See Also

[gauge\(\)](#), [making_gauge\(\)](#), [method_gauge\(\)](#), [class_stat_infer](#)

auto_predict

Automatically predict from a statistical result

Description

auto_predict() is the protocol generic for producing predictions from result objects produced by fn in baseline() and variant(). It is called automatically by predict() when the result stored in cld_exec@data is a class_stat_infer subclass.

Usage

```
auto_predict(x, new_data = NULL, ...)
```

Arguments

x	A class_stat_infer subclass object, typically cld_exec@data.
new_data	A data frame. NULL defaults to the training data.
...	Passed to the dispatched method. Methods typically accept a new_data argument (a data frame; NULL defaults to the training data) plus any method-specific options — see the individual result class's documentation (e.g. ?class_lm_object) for what it accepts.

Details

Register a method on your output class to participate:

```
S7::method(auto_predict, my_model_result) = function(x, new_data = NULL, ...) {
  tibble::tibble(.pred = ...)
}
```

A variant whose fn returns the same result class as baseline inherits auto_predict() for free via S7's parent chain.

Value

A data frame with at minimum a `.pred` column.

See Also

[predict\(\)](#), [making_predict\(\)](#), [method_predict\(\)](#), [class_stat_infer](#)

auto_tidy	<i>Automatically tidy a statistical result</i>
-----------	--

Description

`auto_tidy()` is the protocol generic for tidying result objects produced by `fn` in [baseline\(\)](#) and [variant\(\)](#). It is called automatically by [tidy\(\)](#) when the result stored in `cld_exec@data` is a [class_stat_infer](#) subclass.

Usage

```
auto_tidy(x, ...)
```

Arguments

`x` A [class_stat_infer](#) subclass object, typically `cld_exec@data`.
`...` Currently unused. Passed to the dispatched method.

Details

Register a method on your output class to participate in the protocol:

```
example_out = S7::new_class("example_out", parent = class_stat_infer)
```

```
S7::method(auto_tidy, example_out) = function(x, ...) {
  tibble::tibble(...)
}
```

When a variant's `fn` returns the same output class as `baseline`, it inherits `auto_tidy()` automatically via `S7`'s parent chain. When it returns a subclass, it can override selectively:

```
new_boot_class = S7::new_class("new_boot_class", parent = example_out)
```

```
# override only for boot
# everything else inherited from `example_out`
S7::method(auto_tidy, new_boot_class) = function(x, ...) {
  tibble::tibble(...)
}
```

If no `auto_tidy()` method is found and no [making_tidy\(\)](#) entry exists, [tidy\(\)](#) falls back to an informative error.

Value

A tibble.

See Also

[tidy\(\)](#), [making_tidy\(\)](#), [method_tidy\(\)](#), [class_stat_infer](#)

baseline	<i>Declare the canonical implementation of a test or model</i>
----------	--

Description

`baseline()` declares the default implementation of a statistical procedure. It is always the default and is the only implementation reachable on the eager path.

Usage

```
baseline(fn, print = NULL, claim_parser = NULL)
```

Arguments

fn	A function whose first argument must be <code>.proc</code>, the processed model output from model_processor(). The keys available on <code>.proc</code> depend on the variable mapper <code><var_id></code> used: • <code>x_by</code>: <code>\$x_data</code>, <code>\$group_data</code> • <code>rel</code>: <code>\$x_data</code>, <code>\$resp_data</code> • <code>pairwise</code>: <code>\$var_names</code>, <code>\$pairs</code>, <code>\$data</code> • <code>formula</code>: <code>\$data</code>, <code>\$vars</code>, <code>\$formula</code> Try run this to explore the structure: <code>names(model_processor(<var_id>, <data>))</code>. Additional named arguments are user-supplied statistical parameters (e.g. <code>.mu</code>, <code>.ci</code>). See model_processor() for the full <code>.proc</code> schema per model type. <pre>baseline(fn = function(.proc, .mu = 0, .ci = 0.95) { # ... <your-own-class>(...) # return a class_stat_infer subclass })</pre> When <code>fn</code> returns a class_stat_infer subclass, auto_tidy() and future <code>auto_*</code>() generics dispatch automatically on the result. Otherwise, register a tidy method via making_tidy().
print	A function with signature <code>function(x, ...)</code> for formatting the result. <code>x</code> is a <code>cld_exec</code> object — read your result from <code>x@data</code>. <code>NULL</code> falls back to <code>print(x@data)</code>.
claim_parser	A map_claim() object that maps a <code>null_claim</code> to named arguments injected into <code>fn</code> alongside <code>.proc</code>. <code>NULL</code> (the default) if this implementation does not support state_null().

Value

A baseline S7 object.

See Also

[variant\(\)](#), [agendas\(\)](#), [stat_define\(\)](#), [model_processor\(\)](#), [map_claim\(\)](#), [class_stat_infer](#), [auto_tidy\(\)](#)

claim-vars-validators	<i>Validate hypothesis parameter references against a model's declared variables</i>
-----------------------	--

Description

An S7 generic dispatched on the variable mapper `<var_id>` class. Called automatically inside [state_null\(\)](#) after the compatible-param guard. Implement a method for any new [var_id](#) subclass you define.

Usage

```
validate_claim_vars(var_id, processed, claims, ...)

check_param_nodes(claims, x_vars, by_vars)

validate_one_param_node(node, ...)

check_x_and_given(x_quo, given_quo, x_vars, by_vars, cls_name)
```

Arguments

<code>var_id</code>	A <var_id> object (usually carried by define_model()).
<code>processed</code>	The processed list from <code>lazy@processed</code> , as returned by model_processor() .
<code>claims</code>	A <code>null_claim</code> object.
<code>...</code>	Currently unused.
<code>x_vars</code>	A character vector of declared x-variable names, or NULL to skip x-variable validation.
<code>by_vars</code>	A character vector of declared grouping variable names, or NULL to skip grouping variable validation.
<code>node</code>	A <code>param_obj</code> instance.
<code>x_quo</code>	A quosure holding the x slot value, or NULL.
<code>given_quo</code>	A quosure holding the given slot value, or NULL.
<code>cls_name</code>	A string naming the param class, used in error messages.

Details

`check_param_nodes()` is the shared walker used by all built-in `validate_claim_vars()` methods. It collects every `param_obj` node, runs `validate_one_param_node()` on each, deduplicates errors, and aborts with a consolidated message. Call this inside your own `validate_claim_vars()` method rather than re-implementing the walking and accumulation logic.

`validate_one_param_node()` is an S7 generic dispatched on `param_obj`. Returns a character vector of error strings — empty if valid. The default method on `param_obj` returns `character(0)`, so unknown subclasses pass through safely. Implement a method for any new `param_obj` subclass whose slots should be checked against the model's declared variables.

`check_x_and_given()` is the standard building block for `validate_one_param_node()` methods on `param_obj` subclasses that follow the `MU(x, given)` slot convention. It checks an `x` quosure against `x_vars` and a given quosure against `by_vars`.

Value

`invisible(NULL)`, or aborts with a consolidated error.

Implementing a new method

By default, unknown variable mapper `<var_id>` subclasses pass through without validation. To add validation for a new `var_id` subclass, implement a method and delegate to `check_param_nodes()`:

```
S7::method(validate_claim_vars, <var_id>) = function(var_id, processed, claims) {
  check_param_nodes(
    claims,
    x_vars = names(processed$x_data),
    by_vars = names(processed$group_data)
  )
}
```

Implementing `validate_one_param_node` for a new param class

For a subclass with `x` and `given` slots, delegate to `check_x_and_given()`:

```
S7::method(validate_one_param_node, MY_PARAM) = function(node, x_vars, by_vars) {
  check_x_and_given(node@x, node@given, x_vars, by_vars, "MY_PARAM")
}
```

For a subclass with two variable slots (like `RHO()`):

```
S7::method(validate_one_param_node, MY_PARAM) = function(node, x_vars, by_vars) {
  errors = character(0)
  for (slot_quo in list(node@x, node@y)) {
    lbl = rlang::as_label(slot_quo)
    if (!is.null(x_vars) && !lbl %in% x_vars) {
      errors = c(errors, cli::format_inline(
        "Unknown variable {.val {lbl}} in {.cls MY_PARAM}. ",
        "Model declares x-variable{?s}: {.and {.val {x_vars}}}."
      ))
    }
  }
}
```

```

    ))
  }
}
errors
}
```

See Also

[state_null\(\)](#), [MU\(\)](#), [RHO\(\)](#), [PI\(\)](#), [param_obj\(\)](#), [var_id\(\)](#)

claim_contrast_coefs	<i>Extract contrast coefficients from a null claim</i>
----------------------	--

Description

Decomposes the hypothesis into a named numeric vector of coefficients, one per `param_obj` term, plus the hypothesized scalar value and operator.

Usage

```
claim_contrast_coefs(claim, filter = NULL)
```

Arguments

<code>claim</code>	A <code>null_claim</code> object.
<code>filter</code>	Optional. The name of a conditioning slot (e.g. "given") that every <code>param_obj</code> term must supply. If a term's class declares that slot as an optional constructor argument (default <code>NULL</code>) but the slot is unset on the node, an error is raised. If <code>NULL</code> (the default), the slot to enforce is auto-detected per term by inspecting its class constructor's optional formals (those defaulting to <code>NULL</code>) and requiring the first such slot found.

Value

A list with fields `coefs`, `scalar`, and `op`.

claim_scalar

Extract a scalar hypothesis value from a null claim

Description

Rearranges a hypothesis of the form $c * \text{PARAM} + d == \text{scalar}$ by moving all numeric terms to the right-hand side. Unlike [claim_contrast_coefs\(\)](#), this does not require or validate a linear combination — it is suitable for single-parameter claims involving any [param_obj](#) subclass (e.g. [MU\(\)](#), [PI\(\)](#), [RHO\(\)](#)).

Usage

```
claim_scalar(claim, solve_coef = FALSE)
```

Arguments

claim	A null_claim object.
solve_coef	Logical. If TRUE, divides the scalar by the parameter's coefficient c, fully solving for the parameter value $(\text{scalar} - d) / c$. Errors if $c == 0$. If FALSE, returns $\text{scalar} - d$ only, leaving c on the parameter side. Default FALSE.

Value

A list with fields:

coefs Named numeric vector of length 1: the coefficient c on the parameter term.

scalar Numeric. The resolved scalar value after rearrangement.

op Character. The (possibly flipped) relational operator.

See Also

[claim_contrast_coefs\(\)](#)

class_corr_two

Structured result container for two-sample t-tests

Description

An S7 class produced by [COR_TEST](#) using [rel\(\)](#) and `<formula>` as the variable mapper `<var_id>`. Not constructed manually, use the "grammar interface" instead.

Inherits from [class_stat_infer](#), so [auto_tidy\(\)](#) dispatches on it automatically. Downstream packages can use it as a parent in `S7::new_class()`.

Details

Slots (populated automatically by [COR_TEST](#)):

- ind_vars: name of the independent variables.
- resp_vars: name of the response / dependent variables.
- estimate: the estimated correlation coefficient.
- statistic: t-statistic.
- df: degrees of freedom.
- p_val: p-value.
- lower_ci: lower confidence bound.
- upper_ci: upper confidence bound.
- ci_level: confidence level, e.g. 0.95.

Shared by variants

Both [rel\(\)](#) and `<formula>`'s default (base) return a `class_corr_two`, so different models shares both [auto_tidy\(\)](#) and [print\(\)](#) for free.

See Also

[COR_TEST](#), [auto_tidy\(\)](#), [class_stat_infer](#)

class_glm_object

Structured result container for GLM fits

Description

An S7 class produced by [GLM](#) pipelines. Not constructed manually — use `define_model()` |> `prepare_model(GLM)` |> `conclude()` instead.

Inherits from [anova_able](#), so it participates in [anova\(\)](#) directly. Downstream packages can use it as a parent in `S7::new_class()`.

Details

Constructor arguments (populated automatically by [GLM](#)):

- terms: model terms object.
- df_residual: residual degrees of freedom.
- deviance: scalar deviance.
- dispersion: scalar dispersion parameter.
- family: string naming the error family, e.g. "binomial".
- link: string naming the link function, e.g. "logit".
- null_deviance: scalar deviance of the intercept-only model.

- `aic`: scalar AIC.
- `logLik`: scalar log-likelihood of the fitted model.
- `null_logLik`: scalar log-likelihood of the intercept-only model.
- `beta`: named numeric vector of coefficient estimates.
- `std_beta`: named numeric vector of coefficient standard errors.
- `actual`: numeric vector of the original values on the response scale.
- `fitted`: numeric vector of fitted values on the response scale.
- `vcov`: variance-covariance matrix of the coefficients, e.g. `stats::vcov(fit)`. Required for `predict()` with interval.
- `x_mat`: model matrix stored as a flat numeric vector via `as.numeric(stats::model.matrix(fit))`. Required for `predict()`.
- `x_levels`: factor levels used when fitting, via `stats::getXlevels(fit$terms, stats::model.frame(fit))`. Required for `predict()` on new data with factor predictors.

The following are computed automatically and do not need to be supplied:

- `statistic`: per-coefficient test statistics (`beta / std_beta`).
- `p_value`: per-coefficient two-sided p-values. Uses a z-test when family is "binomial" or "poisson" (fixed dispersion), and a t-test against `df_residual` otherwise (estimated dispersion).
- `coefficients`: tibble with columns `term`, `estimate`, `std_error`, `statistic`, `p_value`.
- `fit_summary`: tibble with columns `family`, `link`, `null_deviance`, `deviance`, `df_residual`, `aic`, `n_obs`.

Value

An S7 object of class `glm_object` holding the fitted GLM's terms, coefficients, dispersion, and family information. Not constructed manually; populated internally by [GLM](#).

`predict()` arguments

`predict()` on a `class_glm_object` accepts:

- `new_data`: A data frame of new predictors. NULL (the default) returns fitted values and response-based truth for the training data.
- `type`: One of "response" (default, back-transformed through the inverse link) or "link" (linear predictor scale).
- `interval`: One of "none" (default) or "confidence". Prediction intervals are not available, since GLMs have no closed-form analogue of OLS prediction error.
- `level`: Confidence level for the interval. Default 0.95.

See Also

[anova_able](#), [GLM](#)

Examples

```
# Inheriting from class_glm_object in a downstream package:
my_glm = S7::new_class(
  "my_glm",
  parent = class_glm_object
)

# Populating class_glm_object from a fitted glm (as done internally):
fit = glm(am ~ wt + hp, data = mtcars, family = binomial())
s = summary(fit)
fam = fit$family$family

obj = class_glm_object(
  terms = fit$terms,
  df_residual = fit$df.residual,
  deviance = fit$deviance,
  dispersion = if (fam %in% c("binomial", "poisson")) 1 else s$dispersion,
  family = fam,
  link = fit$family$link,
  null_deviance = fit$null.deviance,
  aic = fit$aic,
  beta = coef(s)[, 1],
  std_beta = coef(s)[, 2],
  actual = unname(fit$y),
  fitted = unname(fit$fitted.values),
  vcov = vcov(fit),
  x_mat = as.numeric(model.matrix(fit)),
  x_levels = .getXlevels(fit$terms, model.frame(fit))
)

obj@coefficients
obj@fit_summary
```

class_lm_object

Structured result container for linear model fits

Description

An S7 class produced by [LINEAR_REG](#) pipelines. Not constructed manually — use `define_model()` `|> prepare_model(LINEAR_REG)` `|> conclude()` instead.

Inherits from [anova_able](#), so it participates in `anova()` directly. Downstream packages can use it as a parent in `S7::new_class()`.

Details

Constructor arguments (populated automatically by [LINEAR_REG](#)):

- `terms`: model terms object.

- `df_residual`: residual degrees of freedom.
- `deviance`: residual sum of squares (RSS).
- `dispersion`: mean squared error (MSE), i.e. `rss / df_residual`.
- `family`: always "gaussian" for OLS.
- `fitted`: numeric vector of fitted values.
- `residuals`: numeric vector of model residuals.
- `beta`: named numeric vector of coefficient estimates.
- `std_beta`: named numeric vector of coefficient standard errors.
- `x_mat`: model matrix stored as a flat numeric vector via `as.numeric(stats::model.matrix(fit))`. Required for single-model `anova()`; omit only if Type I ANOVA is not needed.

The following are computed automatically and do not need to be supplied:

- `statistic`: per-coefficient t-statistics (`beta / std_beta`).
- `p_value`: per-coefficient two-sided p-values.
- `coefficients`: tibble with columns `term`, `estimate`, `std_error`, `statistic`, `p_value`.
- `fit_summary`: two-column tibble (`statistic`, `value`) with R-squared, adjusted R-squared, sigma, n, residual df, F-statistic, `df1`, `df2`, and F p-value.

Value

An S7 object of class `lm_object` holding the fitted model's terms, coefficients, residuals, and dispersion. Not constructed manually; populated internally by the linear regression pipeline.

anova() protocol

`class_lm_object` supports two `anova()` modes:

- **Single model** — Type I (sequential) ANOVA. Each term is tested against the model containing all preceding terms. Requires `x_mat` to be populated.
- **Multiple models** — incremental F-test or LRT across nested models. Uses `@deviance` and `@df_residual` only; `x_mat` is not needed.

predict() arguments

`predict()` on a `class_lm_object` accepts:

- `new_data`: A data frame of new predictors. NULL (the default) returns fitted values and residual-based truth for the training data.
- `interval`: One of "none" (default), "confidence", or "prediction". Both are available for every `class_lm_object`, since family is always "gaussian" for OLS fits.
- `level`: Confidence level for the interval. Default 0.95.

See Also

`anova()`, `LINEAR_REG`

Examples

```
# Inheriting from class_lm_object in a downstream package:
my_lm = S7::new_class(
  "my_lm",
  parent = statim::class_lm_object
)

# Populating class_lm_object from a fitted lm (as done internally):
fit = lm(dist ~ speed, data = cars)
coef_tbl = summary(fit)$coefficients
rss = sum(fit$residuals^2)
df_res = fit$df.residual

obj = class_lm_object(
  terms = fit$terms,
  fitted = unname(fit$fitted.values),
  residuals = unname(fit$residuals),
  beta = coef_tbl[, 1],
  std_beta = coef_tbl[, 2],
  df_residual = df_res,
  deviance = rss,
  dispersion = rss / df_res,
  family = "gaussian",
  x_mat = as.numeric(stats::model.matrix(fit))
)

obj@coefficients
obj@fit_summary
```

class_p_test

Structured result container for proportion tests

Description

An S7 class produced by [P_TEST](#) pipelines using [prop\(\)](#) as the model ID. Not constructed manually — use the pipeline instead.

Inherits from [class_stat_infer](#), so [auto_tidy\(\)](#) dispatches on it automatically. Downstream packages can use it as a parent in `S7::new_class()`.

Details

Slots (populated automatically by [P_TEST](#)):

- x: number of successes (input).
- n: number of trials (input).
- estimate: observed proportion (x / n).

- **statistic**: test statistic. The count x for the default binomial; the chi-squared value for "prop".
- **p_val**: p-value.
- **lower_ci**: lower confidence bound.
- **upper_ci**: upper confidence bound.
- **ci_level**: confidence level, e.g. 0.95.

Value

An S7 object of class `p_test`, with the properties listed in Details. Not constructed manually; returned by [P_TEST](#) pipelines.

Shared by variants

Both default and prop return a `class_p_test`, so [auto_tidy\(\)](#) and [print\(\)](#) are inherited by prop for free.

See Also

[P_TEST](#), [auto_tidy\(\)](#), [class_stat_infer](#)

class_stat_infer	<i>Base class for all statistical result objects</i>
------------------	--

Description

`class_stat_infer` is the base abstract S7 class for all result objects returned by `fn` in [baseline\(\)](#) and [variant\(\)](#). Concrete result classes like [class_lm_object](#) inherit from it.

Usage

```
class_stat_infer()
```

Details

Inheriting from `class_stat_infer` is the contract that enables automatic dispatch for [auto_tidy\(\)](#), and future [auto_plot\(\)](#) and [auto_export\(\)](#) generics, without any manual registration via [making_tidy\(\)](#).

Value

An S7 abstract class generator. `class_stat_infer` cannot be instantiated directly, so calling it raises an error. It exists only as a parent class for the concrete result classes described in the Class hierarchy section above.

Protocol

Inheriting from `class_stat_infer` is the contract that enables automatic dispatch for `auto_tidy()`, and future `auto_plot()` and `auto_export()` generics, without any manual registration via `making_tidy()`.

When `fn` in `baseline()` or `variant()` returns a `class_stat_infer` subclass, `tidy()` calls `auto_tidy()` on it automatically. Register a method on your result class to participate:

```
example_out = S7::new_class("example_out", parent = class_stat_infer)

S7::method(auto_tidy, example_out) = function(x, ...) {
  # return something
}
```

Variant inheritance

A variant whose `fn` returns the same result class as `baseline` inherits all `auto_*`() methods for free via `S7`'s parent chain. A variant that returns a subclass overrides only what it needs — everything else inherits automatically:

```
my_result_boot = S7::new_class("my_result_boot", parent = my_result)

# only auto_tidy() differs
# all other auto_*( ) inherited from my_result
S7::method(auto_tidy, my_result_boot) = function(x, ...) {
  tibble::tibble(...)
}
```

Class hierarchy

The built-in hierarchy is:

```
class_stat_infer
|-- anova_able
|   |-- class_lm_object
|-- <your-own-output-class>
|   |-- <your-own-subclass>
```

Downstream packages can extend the hierarchy further by using any `class_stat_infer` subclass as a parent in `S7::new_class()`.

See Also

[baseline\(\)](#), [variant\(\)](#), [auto_tidy\(\)](#), [class_lm_object](#)

`class_ttest_one`*Structured result container for one-sample t-tests*

Description

An S7 class produced by `T_TEST` pipelines using `on()` as the variable mapper `<var_id>`. Not constructed manually — use the pipeline instead.

Inherits from `class_stat_infer`, so `auto_tidy()` dispatches on it automatically. Downstream packages can use it as a parent in `S7::new_class()`.

Details

Slots (populated automatically by `T_TEST`):

- `term`: name of the tested variable.
- `estimate`: sample mean.
- `true_mu`: hypothesized mean as written in the claim. Falls back to `.mu` when no `state_null()` claim is supplied.
- `statistic`: t-statistic.
- `p_val`: p-value.
- `lower_ci`: lower confidence bound.
- `upper_ci`: upper confidence bound.
- `ci_level`: confidence level, e.g. `0.95`.

Value

An S7 object of class `ttest_one`, with the properties listed in Details. Not constructed manually; returned by `T_TEST` pipelines.

Shared by variants

Both base and multi return a `class_ttest_one`, so `auto_tidy()` and `print()` are inherited by multi for free.

See Also

`T_TEST`, `ttest-on`, `auto_tidy()`, `class_stat_infer`

class_ttest_pairwise *Structured result container for pairwise t-tests*

Description

An S7 class produced by `T_TEST` pipelines using `pairwise()` as the variable mapper `<var_id>`. Not constructed manually — use the pipeline instead.

Inherits from `class_stat_infer`, so `auto_tidy()` dispatches on it automatically. Downstream packages can use it as a parent in `S7::new_class()`.

Details

Slots (populated automatically by `T_TEST`):

- `var1`: first variable in each pair.
- `var2`: second variable in each pair.
- `est`: mean difference per pair (or sample mean for one-sample mode).
- `df`: degrees of freedom per pair.
- `t_stat`: t-statistic per pair.
- `p_value`: p-value per pair.
- `method_name`: scalar string describing the test method, taken directly from `stats::t.test()`. Must be length 1 — all pairs must share the same method.

Value

An S7 object of class `ttest_pairwise`, with the properties listed in Details. Not constructed manually; returned by `T_TEST` pipelines using `pairwise()`.

One-sample mode

When `pairwise()` uses `direction = "eq"`, `var1` and `var2` are identical (each variable tested against itself). `print()` detects this and renders a diagonal-only matrix.

See Also

`T_TEST`, `ttest-pairwise`, `auto_tidy()`, `class_stat_infer`

class_ttest_two	<i>Structured result container for two-sample t-tests</i>
-----------------	---

Description

An S7 class produced by [T_TEST](#) pipelines using [x_by\(\)](#) as the variable mapper <var_id>. Not constructed manually — use the pipeline instead.

Inherits from [class_stat_infer](#), so [auto_tidy\(\)](#) dispatches on it automatically. Downstream packages can use it as a parent in `S7::new_class()`.

Details

Slots (populated automatically by [T_TEST](#)):

- group: name of the grouping variable.
- estimate: mean difference (or linear contrast estimate).
- t_stat: t-statistic.
- df: degrees of freedom.
- p_val: p-value.
- lower_ci: lower confidence bound.
- upper_ci: upper confidence bound.
- ci_level: confidence level, e.g. 0.95.

Value

An S7 object of class `ttest_two`, with the properties listed in Details. Not constructed manually; returned by [T_TEST](#) pipelines.

Shared by variants

Both the default (base) and contrast return a `class_ttest_two`, so [auto_tidy\(\)](#) and [print\(\)](#) are inherited by contrast for free.

See Also

[T_TEST](#), [auto_tidy\(\)](#), [class_stat_infer](#)

class_var_inform	<i>Output class for Variable Mapper metadata</i>
------------------	--

Description

class_var_inform is the S7 output class returned by `var_id_info()`. model_type is derived automatically from the stored var_id object. All other properties default to empty / unknown values, which are filled in by registered `var_id_info()` methods for known subclasses.

Value

An S7 object of class var_inform holding the Variable Mapper's type, arguments, extracted variables, and registration status. Returned by `var_id_info()`, not constructed manually.

conclude	<i>Execute a lazy pipeline</i>
----------	--------------------------------

Description

conclude() is the terminal step of the pipeline. It resolves the method variant, runs the implementation, and returns a cld_exec S7 object.

Usage

```
conclude(.x, ...)
```

Arguments

.x	A test_lazy or model_lazy object produced by <code>prepare_test()</code> or <code>prepare_model()</code> (optionally followed by <code>via()</code>).
...	Currently unused.

Value

A cld_exec S7 object with the following slots:

@data The raw return value of the fn defined in `baseline()` or `variant()`. Its structure depends on the implementation — see the documentation of the stat function (e.g. ?T_TEST) for what to expect.

@cld_meta A list of pipeline metadata:

\$var_id The Variable Mapper object passed to `define_model()`.

\$processed The processed model output from `model_processor()`. The same object received as .proc inside the fn.

\$stat_name The human-readable test or model name.

\$method The variant name used. "default" when no `via()` was called.

\$data_name The name of the data frame, if resolvable.

Writing print functions

The print argument of `baseline()` and `variant()` receives a `cld_exec` object as `x`. Read your output from `x@data`:

```
baseline(
  fn = function(.proc, .mu = 0) { ... },
  print = function(x, ...) {
    dat = x@data
    # render dat
    invisible(x)
  }
)
```

Otherwise, when the base `S7` class dispatches `print()` elsewhere, it is inherited without writing print from `baseline()` / `variant()`

Writing tidy functions

Prefer implementing `auto_tidy()` on your result class when `fn` returns a `class_stat_infer` subclass. Use `making_tidy()` only when `fn` intentionally returns a non-`class_stat_infer` object.

For example:

```
making_tidy(T_TEST, x_by) %<-% method_tidy(
  default = function(.x, ...) {
    dat = .x@data
    # return a tibble
  }
)
```

See Also

`prepare_test()`, `prepare_model()`, `via()`, `model_processor()`, `class_stat_infer`, `auto_tidy()`

Examples

```
sleep |>
  define_model(x_by(extra, group)) |>
  prepare_test(T_TEST) |>
  conclude()
```

```
sleep |>
  define_model(x_by(extra, group)) |>
  prepare_test(T_TEST) |>
  via("boot", n = 2000) |>
  conclude()
```

```
mtcars |>
  define_model(rel(mpg, wt)) |>
  prepare_model(LINEAR_REG) |>
```

conclude()

COR_TEST	<i>Correlation Test</i>
----------	-------------------------

Description

COR_TEST() performs a correlation test for one-to-one variable relationships. If COR_TEST is supplied within the lazy-loaded pipeline, supply COR_TEST as a function i.e. prepare_test(.test = COR_TEST) call.

Usage

COR_TEST(.var_id = NULL, .data = NULL, ...)

Arguments

- | | |
|---------|---|
| .var_id | A variable mapper <var_id> for COR_TEST(), e.g. rel() . When supplied, the test executes immediately. |
| .data | A data frame. Only used on the standalone path. |
| ... | Additional arguments passed to the implementation. See the Arguments section of each implementation page. |

Value

A cld_exec object (in [conclude\(\)](#)), or a test_spec object when .var_id = NULL. The default correlation test class for most paths is [class_corr_two](#).

Supported variable mapper <var_id>s

Each variable mapper <var_id> routes to a separate implementation. See the linked pages for full argument lists, variants, and correlation test class details:

- [rel\(\)](#): one-to-one correlation test. See details from [cortest-rel](#).
- <formula>: one-to-many correlation test. See details from [cortest-formula](#).

See Also

[cortest-rel](#), [cortest-formula](#) for per-implementation details. [class_corr_two](#) for correlation test class slots. [via\(\)](#), [state_null\(\)](#), [conclude\(\)](#), [auto_tidy\(\)](#).

Examples

```
# eager
COR_TEST(rel(speed, dist), cars)

# grammatical syntax
cars |>
  define_model(rel(speed, dist)) |>
  prepare_test(COR_TEST) |>
  conclude()

cars |>
  define_model(speed ~ dist) |>
  prepare_test(COR_TEST) |>
  conclude()

# Spearman
suppressWarnings({
  cars |>
    define_model(rel(speed, dist)) |>
    prepare_test(COR_TEST) |>
    via("spearman") |>
    conclude()
})

# Custom Hypothesis Expression
cars |>
  define_model(rel(speed, dist)) |>
  prepare_test(COR_TEST) |>
  state_null(RHO(speed, dist) >= 0.8) |>
  conclude()
```

display

Display individual results

Description

`display()` prints the first `n` concluded models from an abstract S7 class, e.g. `multi_exec`, in full. Useful when `conclude()` has been called on a `write_models()` pipeline and the default compressed print is not enough.

Usage

```
display(x, n = 3L, ...)
```

Arguments

<code>x</code>	An object yield by <code>conclude()</code> .
<code>n</code>	A positive integer. The number of models' output to display. Defaults to 3.
<code>...</code>	Currently unused.

Value

`x`, invisibly.

See Also

`write_models()`, `conclude()`, `tidy()`

Examples

```
LifeCycleSavings |>
  write_models(
    f1 = sr ~ 1,
    f2 = sr ~ pop15,
    f3 = sr ~ pop15 + pop75,
    f4 = sr ~ pop15 + pop75 + dpi,
    f5 = sr ~ pop15 + pop75 + dpi + ddpi
  ) |>
  prepare_model(LINEAR_REG) |>
  conclude() |>
  display(2)
```

gauge

Effect size for a concluded statistical result

Description

`gauge()` reports the standardized magnitude of an effect — Cohen's *d*, partial eta-squared, odds ratio, and similar quantities — as distinct from `tidy()`'s raw estimates and the *p*-value's significance verdict.

Usage

```
gauge(object, ...)
```

Arguments

`object` A `cld_exec` object produced by `conclude()`.
`...` Passed to the dispatched method.

Value

A tibble with `metric` and `value` columns, one row per effect-size quantity reported by the underlying result class.

Dispatch

Same two paths as `tidy()/glance()/predict()`:

Path 1: `auto_gauge()` (**preferred**). Called directly when `cld_exec@data` is a `class_stat_infer` subclass.

Path 2: `making_gauge()` **registry (escape hatch)**. Used when a variant's fn intentionally returns a non-`class_stat_infer` object.

See Also

`conclude()`, `auto_gauge()`, `making_gauge()`

GLM

Generalized linear model

Description

A modified GLM for `{statim}` pipeline passed through `stats::glm()`.

Usage

```
GLM(.var_id = NULL, .data = NULL, ...)
```

Arguments

<code>.var_id</code>	A variable mapper <code><var_id></code> from <code>define_model()</code> , or <code>NULL</code> to return a <code>model_spec</code> for use in <code>prepare_model()</code> .
<code>.data</code>	A data frame. Used when <code>.var_id</code> is supplied directly.
<code>...</code>	Additional arguments passed to <code>stats::glm()</code> .

Details

Additional arguments are passed to `stats::glm()`. The most important is `family`, which controls the error distribution and link function (e.g. `stats::binomial()`, `stats::poisson()`). Defaults to `stats::gaussian()` when omitted.

Value

A `cld_exec` object in a `class_glm_object`, or a `model_spec` when `.var_id = NULL`.

Examples

```
# logistic regression
mtcars |>
  define_model(am ~ wt + hp) |>
  prepare_model(GLM) |>
  update(family = binomial()) |>
  conclude()

# model comparison via anova()
mod1 = mtcars |>
  define_model(am ~ 1) |>
  prepare_model(GLM) |>
  update(family = binomial()) |>
  conclude()
mod2 = mtcars |>
  define_model(am ~ wt) |>
  prepare_model(GLM) |>
  update(family = binomial()) |>
  conclude()
mod3 = mtcars |>
  define_model(am ~ wt + hp) |>
  prepare_model(GLM) |>
  update(family = binomial()) |>
  conclude()

anova(mod1, mod2, mod3)
```

HTEST_FN

Build a hypothesis test function

Description

HTEST_FN() is a developer-interface constructor for user-facing test functions like [T_TEST\(\)](#). It returns a function with a consistent signature that routes to the correct implementation based on the variable mapper <var_id> and method variant.

Usage

```
HTEST_FN(cls, defs, .name)
```

Arguments

cls	A string naming the test class, e.g. "ttest".
defs	A list of stat_define() objects.
.name	A string used as the test title in output.

Value

A function with signature `function(.var_id, .data, ...)`.

See Also

[MODEL_FN\(\)](#), [stat_define\(\)](#), [prepare_test\(\)](#), [via\(\)](#), [conclude\(\)](#)

inlines

Inline multiple expressions in a Variable Mapper

Description

`inlines()` is the multi-expression analogue of `c()` for inline data. Where `c(x1, x2)` selects multiple variables or columns by name from a data frame or the calling environment, `inlines()` accepts raw expressions — vectors, function calls, or any R expression — and evaluates them immediately at model definition time.

Usage

```
inlines(...)
```

Arguments

... Named or unnamed expressions. If named, the supplied name becomes the variable name in the processed output. Unnamed elements are auto-named by their role and position: `xv1`, `xv2`, ... under role "x"; `grp1`, `grp2`, ... under role "group"; `pv1`, `pv2`, ... inside [pairwise\(\)](#). Names can be mixed freely — unnamed elements take an auto-name based on their position regardless of whether other elements are named.

Details

Use `inlines()` when your data does not live in a data frame. For a single inline expression, use [I\(\)](#) instead. Take note that `inlines()` does not return an evaluated value, only a naked and unevaluated expression.

Value

A named list of quosures. Intended for use inside variable mapper `<var_id>` functions such as [x_by\(\)](#) and [rel\(\)](#); not typically called on its own.

See Also

[I\(\)](#) for a single inline expression, [x_by\(\)](#), [rel\(\)](#), [pairwise\(\)](#)

Examples

```
# Named inline expressions – names appear in output
x_by(
  inlines(x1 = rnorm(30), x2 = rnorm(30)),
  I(grp = rep(c("a", "b"), each = 15))
)

# Unnamed – auto-named as xv1, xv2 under role "x"
x_by(
  inlines(rnorm(30), rnorm(30)),
  I(rep(c("a", "b"), each = 15))
)

# Mixed – named elements keep their name, unnamed get auto-names
x_by(
  inlines(x1 = rnorm(30), rnorm(30)),
  I(rep(c("a", "b"), each = 15))
)

# Contrast with c() – selects existing variables by name
x1 = rnorm(30)
x2 = rnorm(30)
grp = rep(c("a", "b"), each = 15)
x_by(c(x1, x2), grp)
```

layout-define-base	<i>Define a layout supplied by a Variable Mapper</i>
--------------------	--

Description

`define_model()` captures a variable mapper `<var_id>` and optional data into a `def_var` object that can be passed into [prepare_test\(\)](#).

Usage

```
define_model(.x, ...)
```

Arguments

<code>.x</code>	A variable mapper <code><var_id></code> object from x_by() , rel() , pairwise() , or a formula. It is also dispatched for a data frame class when using the data-first pipe style.
<code>...</code>	Currently unused.

Details

Two dispatch methods are available depending on how `.x` is supplied:

- **A "Variable Mapper" first:** `.x` is a Variable Mapper or formula. Accepts `data`, a data frame (defaults to `parent.frame()`).
- **DataFrame-first:** `.x` is a data frame. Accepts `to_analyze`, a variable mapper or formula, as the second argument.

Value

A `def_var` S3 object containing `var_id` and processed.

Examples

```
# model-ID first
define_model(x_by(extra, group), sleep)

# data-frame first (pipe-friendly)
sleep |> define_model(x_by(extra, group))
```

LINEAR_REG

Linear regression

Description

Fits an ordinary least squares linear regression model. Accepts `rel()` or a formula as the variable mapper `<var_id>`.

Usage

```
LINEAR_REG(.var_id = NULL, .data = NULL, ...)
```

Arguments

<code>.var_id</code>	A variable mapper <code><var_id></code> from <code>define_model()</code> , or <code>NULL</code> to return a <code>model_spec</code> for use in <code>prepare_model()</code> .
<code>.data</code>	A data frame. Used when <code>.var_id</code> is supplied directly.
<code>...</code>	Currently unused.

Details

The result is an `class_lm_object`, which satisfies the `anova()` protocol and prints coefficients and model fit universally across all engines and variants.

Value

A `cld_exec` object containing a `class_lm_object`, or a `model_spec` when `.var_id = NULL`.

Examples

```
# via rel()
cars |>
  define_model(rel(speed, dist)) |>
  prepare_model(LINEAR_REG) |>
  conclude()

# via formula
cars |>
  define_model(dist ~ speed) |>
  prepare_model(LINEAR_REG) |>
  conclude()

# write_models() pipeline
LifeCycleSavings |>
  write_models(
    f1 = sr ~ 1,
    f2 = sr ~ pop15,
    f3 = sr ~ pop15 + pop75,
    f4 = sr ~ pop15 + pop75 + dpi,
    f5 = sr ~ pop15 + pop75 + dpi + ddpi
  ) |>
  prepare_model(LINEAR_REG) |>
  anova()

# individual conclude(), compare after
mod1 = LifeCycleSavings |> define_model(sr ~ 1) |> prepare_model(LINEAR_REG) |> conclude()
mod2 = LifeCycleSavings |> define_model(sr ~ pop15) |> prepare_model(LINEAR_REG) |> conclude()

anova(mod1, mod2)
```

making_gauge

Declare a gauge method for a stat and model type

Description

making_gauge() is the escape hatch for registering effect-size methods when a variant's fn intentionally returns a non- class_stat_infer object. When fn returns a class_stat_infer subclass, implement auto_gauge() on the result class instead — no registration needed.

Usage

```
making_gauge(obj, model_type)
```

Arguments

obj A stat function built with [HTEST_FN\(\)](#) or [MODEL_FN\(\)](#) (e.g. T_TEST, LINEAR_REG).

model_type An S7 variable mapper <var_id> class, or S7::class_formula.

Value

A making_gauge_call object, consumed by %<-%.

See Also

[auto_gauge\(\)](#), [method_gauge\(\)](#), [class_stat_infer](#)

Examples

```
# Only needed when fn returns a non-class_stat_infer object.
# Prefer implementing auto_gauge() on your result class instead.
making_gauge(T_TEST, x_by) %<-% method_gauge(
  default = function(.x, ...) { ... }
)
```

making_predict

Declare a predict method for a stat and model type

Description

making_predict() is the escape hatch for registering predict methods when a variant's fn intentionally returns a non-class_stat_infer object. When fn returns a class_stat_infer subclass, implement auto_predict() on the result class instead — no registration needed.

Usage

```
making_predict(obj, model_type)
```

Arguments

obj A stat function built with [MODEL_FN\(\)](#) (e.g. LINEAR_REG).
 model_type An S7 variable mapper <var_id> class, or S7::class_formula.

Value

A making_predict_call object, consumed by %<-%.

See Also

[auto_predict\(\)](#), [method_predict\(\)](#), [class_stat_infer](#)

Examples

```
# Only needed when fn returns a non-class_stat_infer object.
# Prefer implementing auto_predict() on your result class instead.
making_predict(LINEAR_REG, S7::class_formula) %<-% method_predict(
  default = function(.x, new_data = NULL, ...) { ... }
)
```

making_tidy

Declare tidy methods for a stat and model type

Description

making_tidy() is the escape hatch for registering tidy methods when a variant's fn intentionally returns a non-class_stat_infer object (plain list, S3, S4, or R6). When fn returns a class_stat_infer subclass, implement auto_tidy() on the result class instead — no registration needed.

Usage

```
making_tidy(obj, model_type)
```

Arguments

obj A stat function built with HTEST_FN() or MODEL_FN() (e.g. T_TEST).
model_type An S7 variable mapper <var_id> class (e.g. x_by, S7::class_formula).

Value

A making_tidy_call object, consumed by %<-%.

See Also

[auto_tidy\(\)](#), [method_tidy\(\)](#), [class_stat_infer](#)

Examples

```
# Only needed when fn returns a non-class_stat_infer object.
# Prefer implementing auto_tidy() on your result class instead.
making_tidy(T_TEST, x_by) %<-% method_tidy(
  default = function(.x, ...) { ... },
  boot = function(.x, ...) { ... }
)
```

map_claim

Build a claim parser from named resolver functions

Description

map_claim() produces a parser function by mapping impl fn formal names to resolver functions. Each resolver receives (claim, processed) and returns the value for its argument. Resolvers that only need claim can simply ignore processed.

Usage

```
map_claim(...)
```

Arguments

... Named resolver functions. Names must match formals of the impl's fn. Each resolver has signature `function(claim, processed)`.

Details

Pass the result as `claim_parser` to [baseline\(\)](#) or [variant\(\)](#). A variant without a `claim_parser` simply does not support [state_null\(\)](#); `conclude()` raises an error if a claim was stated but the active variant has none.

Value

A function of class "map_claim" with signature `function(claim, processed)`.

method_gauge	<i>Declare gauge methods for a stat result</i>
--------------	--

Description

`method_gauge()` is the companion to `making_gauge()`. It collects effect-size functions for the base implementation and named variants, used only when `fn` returns a non-`class_stat_infer` object.

Usage

```
method_gauge(default = NULL, ...)
```

Arguments

`default` A function with signature `function(.x, ...)`, returning a tibble with `metric` and `value` columns. Required.

... Named functions, one per variant. Names must match variant names registered in `agendas()`. Omitted variants fall back to `default` automatically.

Value

A `method_gauge` S7 object.

See Also

[making_gauge\(\)](#), [auto_gauge\(\)](#), [class_stat_infer](#)

method_predict	<i>Declare predict methods for a stat result</i>
----------------	--

Description

`method_predict()` is the companion to `making_predict()`. It collects predict functions for the base implementation and named variants, used only when `fn` returns a non-`class_stat_infer` object.

Usage

```
method_predict(default = NULL, ...)
```

Arguments

<code>default</code>	A function with signature <code>function(.x, new_data = NULL, ...)</code> . Required.
<code>...</code>	Named functions, one per variant. Names must match variant names registered in <code>agendas()</code> . Omitted variants fall back to <code>default</code> automatically.

Value

A `method_predict` S7 object.

See Also

[making_predict\(\)](#), [auto_predict\(\)](#), [class_stat_infer](#)

method_tidy	<i>Declare tidy methods for a stat result</i>
-------------	---

Description

`method_tidy()` is the companion to [making_tidy\(\)](#). It collects tidy functions for the base implementation and named variants, used only when `fn` returns a non-[class_stat_infer](#) object.

Usage

```
method_tidy(default = NULL, ...)
```

Arguments

<code>default</code>	A function with signature <code>function(.x, ...)</code> . Required.
<code>...</code>	Named functions, one per variant. Names must match variant names registered in agendas() . Omitted variants fall back to <code>default</code> automatically.

Value

A `method_tidy` S7 object.

See Also

[making_tidy\(\)](#), [auto_tidy\(\)](#), [class_stat_infer](#)

model-processor

Model evaluator

Description

A function for development use to extract the information in Variable Mappers.

Usage

```
model_processor(var_id, data = NULL, ...)
```

Arguments

<code>var_id</code>	The Variable Mappers to be extracted.
<code>data</code>	Optional. Only passed when a certain data structure (normally it's data frame) is required.
<code>...</code>	Passed through S7 method compatibility.

Details

Methods accept an optional `data` argument — a data frame, or `NULL` to resolve variables from the calling environment.

Value

A named list. The default method returns an empty list; each registered method returns a list shaped for its `var_id` subclass (for example, `x_data/group_data` for [x_by\(\)](#), or `x/n` for [prop\(\)](#)).

MODEL_FN	<i>Build a model inference function</i>
----------	---

Description

MODEL_FN() is a developer-interface constructor for user-facing model functions like LINEAR_REG(). It returns a function that routes to the correct implementation based on the variable mapper <var_id> and method variant.

Usage

```
MODEL_FN(cls, defs, .name)
```

Arguments

cls	A string naming the model class, e.g. "linear_reg".
defs	A list of stat_define() objects.
.name	A string used as the model title in output.

Value

A function with signature `function(.var_id, .data, ...)`.

See Also

[HTEST_FN\(\)](#), [stat_define\(\)](#), [prepare_model\(\)](#), [via\(\)](#), [conclude\(\)](#)

modifying-assignment	<i>Apply a method_tidy to a making_tidy target</i>
----------------------	--

Description

%<-% registers a [method_tidy\(\)](#) into the tidy registry. The left-hand side must be a [making_tidy\(\)](#) call.

Usage

```
lhs %<-% rhs
```

Arguments

lhs	A making_tidy_call object from making_tidy() .
rhs	A method_tidy() object.

Value

NULL invisibly, called for its side effects.

Examples

```
making_tidy(T_TEST, x_by) %<-% method_tidy(
  default = function(.x, ...) { ... },
  boot = function(.x, ...) { ... }
)
```

 MU

Mean of a variable, optionally conditioned on a subgroup

Description

Mean of a variable, optionally conditioned on a subgroup

Usage

```
MU(x, given = NULL)
```

Arguments

x	A bare variable name.
given	An optional filter predicate as a bare expression.

Value

A MU / param_obj S7 object.

Examples

```
MU(extra)
MU(extra, group == "1")
```

null-hyp

*State a null hypothesis in the pipeline***Description**

`state_null()` captures a hypothesis expression and attaches it to a `test_lazy` object.

Usage

```
state_null(.x, ...)
```

Arguments

`.x` A `test_lazy` object from `prepare_test()`.
`...` Currently unused.

Value

The modified `test_lazy` object.

Slots

`expr` A hypothesis expression. It is passed after `prepare_test(...)` to supply the hypothesis expression, e.g. `... |> prepare_test(T_TEST) |> state_null(expr = MU(x) == 0)`

Examples

```
# Using binomial test as a simple example
define_model(prop(45, 100)) |>
  prepare_test(P_TEST) |>
  state_null(2 * PI() == 0.25) |>
  conclude()
```

on

*Specify variables for independent testing***Description**

`on()` creates an `on` Variable Mapper that describes one or more variables to be tested independently. Expressions are captured unevaluated, similar to how `ggplot2::aes()` captures aesthetics.

Usage

```
on(..., .block = NULL)
```

Arguments

- ... Bare variable names, tidyselect helpers (requires data in `define_model()`), or `I(expr)` for inline data.
- .block Optional. The blocking variable identifying experimental units. Accepts a bare name, a tidyselect helper, or `I(expr)` for inline data. When supplied, rows are sorted by this variable before injection into the test function. Required for tests such as "repeated measures ANOVA".

Details

When multiple variables are supplied, the intended test is run once per variable with no pairwise combinations formed — for that, use `pairwise()`.

The optional `.block` argument identifies the experimental unit (e.g. a subject ID column). When supplied, rows in the extracted data are aligned by block before the test is run — this is required for blocked designs such as the Friedman test where row position encodes block identity.

Value

An `on / var_id S7` object.

See Also

`x_by()`, `rel()`, `pairwise()`, `prop()`

Examples

```
# Single variable
on(x)

# Multiple variables – test on independently
on(a, b, c)

# Multiple variables – With a blocking factor
on(pre, post, followup, .block = subject)

# Tidyselect (requires data in define_model())
on(where(is.numeric))
```

P_TEST	<i>Proportion Test</i>
--------	------------------------

Description

`P_TEST()` performs a one-sample proportion test using either an exact binomial test or a normal approximation. If `P_TEST` is supplied within the lazy-loaded pipeline, supply `P_TEST` as a function within i.e. `prepare_test(. test = P_TEST)` call.

Usage

```
P_TEST(.var_id = NULL, .data = NULL, ...)
```

Arguments

<code>.var_id</code>	A registered variable mapper <var_id>, e.g. <code>prop()</code> . When supplied, the test executes immediately.
<code>.data</code>	Unused. Accepted for pipeline consistency.
<code>...</code>	Additional arguments passed to the implementation. See the Arguments and Variants sections below.

Value

A `cld_exec` object, or a `test_spec` when `.var_id = NULL`. The object stored in `cld_exec@data` is a `class_p_test` object.

Arguments

The following arguments are passed via `...` in `P_TEST()` or `via()`:

<code>.p</code>	Numeric. Hypothesized proportion under H_0 , used directly in <code>stats::binom.test()</code> / <code>stats::prop.test()</code> . Default 0.5. When a hypothesis is stated via <code>state_null()</code> with a scaled claim like <code>c * PI()</code> <code>== k</code> , <code>.p</code> is resolved to the solved value <code>k / c</code> , since <code>c * PI() == k</code> and <code>PI() == k / c</code> are the same hypothesis (the binomial likelihood is invariant under this linear reparameterization).
<code>.alt</code>	Direction: "two.sided", "greater", or "less". Default "two.sided".
<code>.ci</code>	Confidence level. Default 0.95.
<code>.true_p</code>	Only meaningful via <code>state_null()</code> . Carries the hypothesis's scalar value <i>as written</i> (unsolved), purely for display in <code>true_p</code> . Default <code>NULL</code> , in which case <code>true_p</code> falls back to <code>.p</code> . Not intended to be set directly by users.

Variants

"prop" Normal approximation via `stats::prop.test()` without continuity correction. Accepts the same `.p`, `.alt`, `.ci` arguments as the default, except with `correct` addition to indicate whether Yates' continuity correction should be applied or not.

Hypothesis claims

Supports `PI()` via `state_null()`:

```
define_model(prop(45, 100)) |>
  prepare_test(P_TEST) |>
  state_null(PI() == 0.5) |>
  conclude()
```

Scaled claims are also supported, e.g. `2 * PI() == 0.3`. The test itself solves for `PI()` (`.p = 0.15`) and runs exactly via `stats::binom.test()` or `stats::prop.test()` — no approximation is introduced by the scaling, since testing `c * PI() == k` is mathematically identical to testing `PI() == k / c`. `true_p` in the printed/tidied output instead shows the scalar as written on the right-hand side of the claim (`0.3`, not the solved `0.15`), so the displayed hypothesis matches what was typed even though the underlying test operates on the solved proportion.

See Also

`prop()`, `class_p_test`, `PI()`, `state_null()`, `via()`, `conclude()`

Examples

```
P_TEST(prop(45, 100))

# piped syntax
define_model(prop(45, 100)) |>
  prepare_test(P_TEST) |>
  conclude()

# normal approximation
define_model(prop(45, 100)) |>
  prepare_test(P_TEST) |>
  via("prop") |>
  conclude()

# hypothesis claim
define_model(prop(45, 100)) |>
  prepare_test(P_TEST) |>
  state_null(PI() == 0.3) |>
  conclude()

# scaled hypothesis claim
define_model(prop(45, 100)) |>
  prepare_test(P_TEST) |>
  state_null(2 * PI() == 0.3) |>
  conclude()
```

pairwise

Define all pairwise variable combinations

Description

`pairwise()` creates a pairwise Variable Mapper from a set of variables, producing all unique variable pairs. Use `direction` to control which pairs are retained. Pairs are filtered by lexicographic (alphabetical) ordering of variable names.

Usage

```
pairwise(..., direction = "lt")
```

Arguments

... Bare variable names, tidyselect helpers (requires data in `define_model()`), or `I(expr)` for inline data.

direction A string controlling which pairs are kept:

- "lt" (default): strict lower triangle, i.e. pairs where $\text{index}(x) < \text{index}(y)$
- "lteq": lower triangle including the diagonal ($x \leq y$)
- "gt": strict upper triangle ($x > y$)
- "gteq": upper triangle including the diagonal ($x \geq y$)
- "eq": diagonal only ($x == y$), i.e. each variable paired with itself
- "neq": all pairs except the diagonal ($x \neq y$)
- "all": all combinations including both directions and the diagonal

Value

A pairwise / var_id S7 object.

Examples

```
pairwise(a, b, c)

# Inline data
pairwise(I(rnorm(30)), I(rnorm(30)), I(rnorm(30)))

# Keep all ordered pairs
pairwise(a, b, c, direction = "all")
```

param_obj

Base class for population parameters

Description

param_obj is the abstract base S7 class for all population parameter objects, analogous to `var_id()`. Concrete subclasses (MU, PI, RHO) inherit from it. The base class is a pure marker — each subclass declares its own properties.

Usage

```
param_obj()
```

Value

An S7 abstract class generator. param_obj cannot be instantiated directly, so calling it raises an error. It exists only as a parent class for the concrete parameter classes (MU, PI, RHO).

PI	<i>Proportion of a variable, optionally conditioned on a subgroup</i>
----	---

Description

Proportion of a variable, optionally conditioned on a subgroup

Usage

```
PI(x, given = NULL)
```

Arguments

x	An empty or a bare variable name.
given	An optional filter predicate as a bare expression.

Value

A PI / param_obj S7 object.

Examples

```
PI()
PI(success)
PI(success, group == "treatment")
```

predict	<i>Predict from a concluded statistical result</i>
---------	--

Description

predict() estimates the response for new or existing rows from a cld_exec object produced by conclude().

Usage

```
predict(object, new_data = NULL, ..., check_type = TRUE)
```

Arguments

object	A cld_exec object produced by conclude().
new_data	A data frame (or subclass, e.g. tibble, data.table). NULL (the default) uses the training data.
...	Passed to the dispatched method.
check_type	Check whether the returned output is a data frame. If TRUE, predict() is enforcing the dispatched returned output to be a data frame. Default is TRUE.

Format

An object of class `S7_external_generic` of length 4.

Value

A data frame (specifically a tibble) with `.pred`, `truth` when a response is available, and `.pred_lower/.pred_upper` when an interval was requested. Always inherits `data.frame`, regardless of which method produced it.

Dispatch

Dispatches on object only, consistent with how `stats::predict` itself works as a single-dispatch S3 generic. `new_data`'s shape is validated at the top of the method body rather than through a second S7 dispatch argument — an S7 method can't reliably distinguish "argument omitted entirely" from "argument explicitly NULL" when layered on top of a legacy S3 generic, so validating inside the body is both simpler and actually correct.

Two paths are tried in order:

Path 1: `auto_predict()` (**preferred**). Called directly when `cld_exec@data` is a `class_stat_infer` subclass.

Path 2: `making_predict()` **registry (escape hatch)**. Used when a variant's `fn` intentionally returns a non-`class_stat_infer` object.

See Also

[conclude\(\)](#), [auto_predict\(\)](#), [making_predict\(\)](#)

```
prepare
```

Prepare a lazy inference pipeline

Description

`prepare()` attaches a spec function produced from `STAT_CONSTRUCTOR()` to a `<def_var>` object and dispatches to `prepare_test()` or `prepare_model()` depending on whether `.fn` returns a `<test_spec>` or a `<model_spec>`. The result is a lazy pipeline object ready for optional recalibration with `via()` before execution with `conclude()`.

Usage

```
prepare(.x, .fn, ...)
```

Arguments

<code>.x</code>	A <code><def_var></code> object from define_model() , or an <code><expanded_model></code> object from write_models() .
<code>.fn</code>	A stat function built with <code>STAT_CONSTRUCTOR()</code> that returns a <code><test_spec></code> (e.g. T_TEST) or a <code><model_spec></code> (e.g. LINEAR_REG).
<code>...</code>	Additional arguments passed to the dispatched <code>prepare_*</code> () function.

Value

A `<test_lazy>` object if `.fn` returns a `<test_spec>`, or a `<model_lazy>` object if `.fn` returns a `<model_spec>`.

See Also

[prepare_test\(\)](#), [prepare_model\(\)](#), [define_model\(\)](#), [via\(\)](#), [conclude\(\)](#)

Examples

```
sleep |>
  define_model(x_by(extra, group)) |>
  prepare(T_TEST) |>
  conclude()
```

```
mtcars |>
  define_model(mpg ~ .) |>
  prepare(LINEAR_REG) |>
  conclude()
```

```
prepare-model
```

Lazily prepare a model inference

Description

`prepare_model()` attaches a model specification to a `<def_var>` object, producing a `<model_lazy>` ready for optional recalibration with [via\(\)](#) before being executed with [conclude\(\)](#).

Usage

```
prepare_model(.x, .model_fn, ...)
```

Arguments

<code>.x</code>	An S7 object extension yielded by, e.g. <code><def_var></code> object from define_model() , or an <code><expanded_model></code> object from write_models() .
<code>.model_fn</code>	A model function such as LINEAR_REG() .
<code>...</code>	Additional arguments passed to methods.

Value

A `<model_lazy>` S7 object.

See Also

[prepare_test\(\)](#), [define_model\(\)](#), [via\(\)](#), [conclude\(\)](#)

Examples

```
mtcars |>
  define_model(rel(mpg, wt)) |>
  prepare_model(LINEAR_REG) |>
  conclude()
```

```
prepare-test
```

```
Lazily prepare a single test
```

Description

`prepare_test()` attaches a test specification to a `<def_var>` object, producing a `test_lazy` ready for optional recalibration with [via\(\)](#) before being executed with [conclude\(\)](#).

Usage

```
prepare_test(.x, .test, ...)
```

Arguments

<code>.x</code>	An S7 object extension yielded by, e.g. <code><def_var></code> object from define_model() , or an <code><expanded_model></code> object from write_models() .
<code>.test</code>	A test function such as T_TEST that carries <code><test_spec></code> objects when called.
<code>...</code>	Additional arguments passed to methods.

Value

A `<test_lazy>` S7 object.

See Also

[define_model\(\)](#), [via\(\)](#), [conclude\(\)](#)

Examples

```
sleep |>
  define_model(x_by(extra, group)) |>
  prepare_test(T_TEST) |>
  conclude()
```

prop	<i>Define a proportion test model</i>
------	---------------------------------------

Description

prop() creates a prop Variable Mapper for proportion tests. Both arguments are scalar constants, and this implies the arguments are expressions that are not captured.

Usage

```
prop(x, n)
```

Arguments

x	Number of successes. A non-negative integer scalar, $x \leq n$.
n	Total number of trials. A positive integer scalar.

Value

A prop / var_id S7 object.

Examples

```
prop(45, 100)
```

purge_stat_defines	<i>Purge all package-scoped stat_define registrations for a package</i>
--------------------	---

Description

Call this from your package's .onUnload() to clean up entries registered via add_stat_define(..., origin = "package").

Usage

```
purge_stat_defines(pkg)
```

Arguments

pkg	A string. The package name, typically the pkgname argument passed to .onUnload().
-----	---

Value

NULL, invisibly.

Examples

```
# In your package's zzz.R:
.onUnload = function(libpath) {
  statim::purge_stat_defines("yourpackage")
}
```

rel

*Describe the relationship between two variables***Description**

`rel()` creates a `rel` Variable Mapper that reads as "relationship between `x` and `resp`". Expressions are captured unevaluated, similar to how `ggplot2::aes()` captures aesthetics.

Usage

```
rel(x, resp)
```

Arguments

<code>x</code>	The predictor variable. Accepts a bare name, a <code>c()</code> of bare names, a <code>tidyselect</code> helper (requires data in <code>define_model()</code>), or <code>I(expr)</code> for inline data.
<code>resp</code>	The response variable. Same rules as <code>x</code> .

Value

A `rel / var_id` S7 object.

Examples

```
rel(speed, dist)
```

RHO

*Population correlation between two variables***Description**

Population correlation between two variables

Usage

```
RHO(x, y)
```

Arguments

x	A bare variable name.
y	A bare variable name.

Value

A RHO / param_obj S7 object.

Examples

```
RHO(speed, dist)
```

stat-infer-definer	<i>Define a statistical procedure implementation</i>
--------------------	--

Description

stat_define() declares a single implementation of a statistical procedure for a given model type. Multiple stat_define objects are passed to [HTEST_FN\(\)](#) or [MODEL_FN\(\)](#) via defs. This is the main extension point for adding new tests or models.

Usage

```
stat_define(model_type = NULL, impl = NULL, compatible_params = list())

test_define(model_type = NULL, impl = NULL, compatible_params = list())

model_infer_define(model_type = NULL, impl = NULL, compatible_params = list())
```

Arguments

model_type	A variable mapper <var_id> class this implementation handles (e.g. x_by, S7::class_formula).
impl	An agendas() object collecting all implementations. The fn of each baseline() and variant() inside receives .proc as its first argument. See baseline() for the expected signature and model_processor() for the keys available on .proc per model type.
compatible_params	A list of S7 param classes (e.g. list(MU, PI)) this implementation accepts in hypothesis claims. An empty list (the default) disables the check entirely. Useful when a test is param-agnostic or the restriction has not yet been declared. Applies to every variant in impl.

Value

A stat_define S7 object.

See Also

[agendas\(\)](#), [baseline\(\)](#), [variant\(\)](#), [model_processor\(\)](#), [HTEST_FN\(\)](#), [MODEL_FN\(\)](#)

STAT_CONSTRUCTOR	<i>Main foundation for inferential statistics</i>
------------------	---

Description

This function is a developer-interface function, a constructor for user-facing test functions like [HTEST_FN\(\)](#). It returns a function with a consistent signature that routes to the correct implementation based on the variable mapper <var_id> and method variant.

Usage

```
STAT_CONSTRUCTOR(cls, defs, .name, spec_class)
```

Arguments

- cls A string naming the test class, e.g. "ttest".
- defs A list of test_define objects declaring the implementations.
- .name A string used as the test title in output.
- spec_class Base class of the type of statistical inference. Must be an S7.

Value

A function with signature `function(.var_id, .data, ...)`.

See Also

[test_define\(\)](#), [prepare_test\(\)](#), [via\(\)](#), [conclude\(\)](#)

T_TEST	<i>T-Test</i>
--------	---------------

Description

T_TEST() performs a t-test for one-sample, two-sample, paired, pairwise, or formula-based comparisons. If T_TEST is supplied within the lazy-loaded pipeline, supply T_TEST as a function within i.e. `prepare_test(.test = T_TEST)` call.

Usage

```
T_TEST(.var_id = NULL, .data = NULL, ...)
```

Arguments

<code>.var_id</code>	A variable mapper <code><var_id></code> from <code>x_by()</code> , <code>pairwise()</code> , or a formula. When supplied, the test executes immediately.
<code>.data</code>	A data frame. Only used on the standalone path.
<code>...</code>	Additional arguments passed to the implementation. See the Arguments by variable mapper section for the full list per path.

Value

A `cld_exec` object (in `conclude()`), a `stat_infer_spec` object, or a `test_spec` when `.var_id = NULL`. Depending on the implementation you wrote, it returns any class. However, by default, some implementations use base `{statim}` S7 classes. For instance:

- `ttest_x_by`, by default, returns a `class_ttest_two` object
- `ttest_pairwise`, by default, returns a `class_ttest_pairwise` object

Supported variable mapper `<var_id>`s

Each variable mapper `<var_id>` routes to a separate implementation. See the linked pages for full argument lists, variants, and result class details:

- `on()`: one-sample t-test. See details from [ttest-on](#).
- `x_by()`: two-sample or paired t-test. See details from [ttest-xby](#).
- `pairwise()`: pairwise t-tests across variables. See details from [ttest-pairwise](#).
- `<formula>`: one-sample and/or two-sample t-test. See details from [ttest-formula](#).

References

- Welch, B. L. (1947). The generalization of "Student's" problem when several different population variances are involved. *Biometrika*, 34(1-2), 28-35. doi:[10.1093/biomet/34.12.28](#)
- Satterthwaite, F. E. (1946). An approximate distribution of estimates of variance components. *Biometrics Bulletin*, 2(6), 110-114. doi:[10.2307/3002019](#)
- Kutner, M. H., Nachtsheim, C. J., Neter, J., & Li, W. (2004). *Applied Linear Statistical Models* (5th ed.). McGraw-Hill/Irwin.

See Also

[ttest-on](#), [ttest-xby](#), [ttest-pairwise](#), [ttest-formula](#) for per-implementation details. [class_ttest_two](#), [class_ttest_pairwise](#) for result class slots. [via\(\)](#), [state_null\(\)](#), [conclude\(\)](#), [auto_tidy\(\)](#).

Examples

```
# eager
T_TEST(x_by(extra, group), sleep)

# pipeline
sleep |>
  define_model(x_by(extra, group)) |>
```

```

      prepare_test(T_TEST) |>
      conclude()

# bootstrap
sleep |>
  define_model(x_by(extra, group)) |>
  prepare_test(T_TEST) |>
  via("boot", n = 2000) |>
  conclude()

# permutation
sleep |>
  define_model(x_by(extra, group)) |>
  prepare_test(T_TEST) |>
  via("permute", n = 2000) |>
  conclude()

# Contrast t-test
# This uses `state_null()` in a higher degree
# This performs Welch-Satterthwaite linear contrast test for t-test
sleep |>
  define_model(x_by(extra, group)) |>
  prepare_test(T_TEST) |>
  state_null(
    2 * MU(extra, group == "1") <= MU(extra, group == "2")
  ) |>
  # Try to obtain 90% of the confidence interval
  via("contrast", .ci = 0.9) |>
  conclude()

# pairwise
iris |>
  define_model(pairwise(Sepal.Length, Sepal.Width, Petal.Length)) |>
  prepare_test(T_TEST) |>
  conclude()

```

tidy

Tidy a concluded statistical result

Description

`tidy()` extracts a tibble of primary results from a `cld_exec` object produced by `conclude()`.

Usage

```
tidy(.x, ...)
```

Arguments

`.x` A `cld_exec` object produced by `conclude()`.
`...` Passed to the dispatched method.

Value

The statistical output in a tibble data frame format.

Dispatch

Two paths are tried in order:

Path 1: `auto_tidy()` (preferred). When `cld_exec@data` is a `class_stat_infer` subclass, `auto_tidy()` is called directly on it. You need no registry, and S7 automatically dispatches on the particular output class, and variants that return the same class inherit the method automatically via the parent chain.

Path 2: `making_tidy()` registry (escape hatch). When `cld_exec@data` is not a `class_stat_infer` subclass, for example, when a variant intentionally returns any data structure e.g. just a plain list, S3, S4, or R6 object (`check_sic_s7 = FALSE`), `tidy()` falls back to the registry populated by `making_tidy()`. If no entry exists there either, an informative error is raised.

See Also

`auto_tidy()`, `making_tidy()`, `method_tidy()`, `class_stat_infer`

Examples

```
mtcars |>
  define_model(mpg ~ .) |>
  prepare_model(LINEAR_REG) |>
  conclude() |>
  tidy()
```

var_id

Base class for Variable Mapper objects

Description

`var_id` is the abstract parent class for all Variable Mapper objects in `{statim}`. Variable Mappers emulate R's formula interface, as they capture variable expressions without evaluating them, describing the structure of a statistical model to be passed into a pipeline.

Details

Concrete subclasses include `x_by()`, `rel()`, and `pairwise()`. You cannot instantiate `var_id` directly; use one of its subclasses.

Value

An S7 abstract class generator. `var_id` cannot be instantiated directly, so calling it raises an error. It exists only as a parent class for the concrete Variable Mapper subclasses (`x_by()`, `rel()`, `pairwise()`, `prop()`).

See Also

`x_by()`, `rel()`, `pairwise()`, `prop()`

var_id_info	<i>Extract metadata from a Variable Mapper</i>
-------------	--

Description

`var_id_info()` extracts a consistent metadata structure from a Variable Mapper object. When `processed` is supplied, variable previews and count-based metadata are included in the result.

Usage

```
var_id_info(.var_id, processed = NULL, ...)
```

Arguments

<code>.var_id</code>	A Variable Mapper object from <code>x_by()</code> , <code>rel()</code> , <code>pairwise()</code> , or a formula.
<code>processed</code>	A named list returned by <code>model_processor()</code> , or <code>NULL</code> . When <code>NULL</code> , count-based fields in <code>other_info</code> and <code>vars</code> are omitted.
<code>...</code>	Currently unused.

Value

A `class_var_inform` S7 object with fields:

`var_id` The original Variable Mapper object.

`model_type` Derived from the class name of `var_id`.

`args` A formatted string summarising the model's arguments. Defaults to "<?>" for unregistered subclasses.

`other_info` A named list of model-type-specific metadata. Empty for unregistered subclasses.

`vars` A list of lists with name and preview fields. Empty for unregistered subclasses or when `processed` is `NULL`.

Examples

```
# without processed – no vars, no counts
var_id_info(x_by(extra, group))

# with processed – includes vars and counts
dm = define_model(x_by(extra, group), sleep)
var_id_info(dm@var_id, dm@processed)
```

variant

Declare an alternative implementation of a test or model

Description

`variant()` declares a named alternative implementation reachable only via [via\(\)](#). Never runs on the eager path.

Usage

```
variant(fn, print = NULL, claim_parser = NULL)
```

Arguments

fn A function whose first argument must be `.proc`, the processed model output from [model_processor\(\)](#). The keys available on `.proc` depend on the variable mapper `<var_id>` used:

- `x_by`: `$x_data`, `$group_data`
- `rel`: `$x_data`, `$resp_data`
- `pairwise`: `$var_names`, `$pairs`, `$data`
- `formula`: `$data`, `$vars`, `$formula`

Try run this to explore the structure: `names(model_processor(<var_id>, <data>))`.

Additional named arguments are user-supplied statistical parameters (e.g. `.mu`, `.ci`). See [model_processor\(\)](#) for the full `.proc` schema per model type.

```
variant(
  fn = function(.proc, n = 1000L, seed = NULL) {
    x = .proc$x_data[[1]]
    group_data = .proc$group_data
    # ...
  }
)
```

A variant whose `fn` returns the same [class_stat_infer](#) subclass as baseline inherits [auto_tidy\(\)](#) and all future `auto_*`() methods automatically via S7's parent chain. A variant returning a subclass can override selectively:

```

# inherits `auto_tidy()` from `new_out` S7 class
variant(fn = function(.proc, ...) { new_out(...) })

# overrides auto_tidy() via subclass
variant(fn = function(.proc, ...) { new_out_boot(...) })

# intentionally plain
variant(fn = function(.proc, ...) { list(...) })

print      A function with signature function(x, ...). x is a cld_exec object. NULL
           falls back to print(x@data).

claim_parser A map\_claim\(\) object that maps a null_claim to named arguments injected
           into fn alongside .proc. NULL (the default) if this variant does not support
           state\_null\(\).

```

Value

A variant S7 object.

See Also

[baseline\(\)](#), [agendas\(\)](#), [via\(\)](#), [model_processor\(\)](#), [map_claim\(\)](#), [class_stat_infer](#), [auto_tidy\(\)](#)

via	<i>Recalibrate the method variant</i>
-----	---------------------------------------

Description

[via\(\)](#) switches a lazy pipeline to an alternative method variant and merges user-supplied arguments with the variant's declared defaults. Works for both `test_lazy` and `model_lazy` pipelines.

Usage

```
via(.x, .method, ...)
```

Arguments

.x	A <code>test_lazy</code> or <code>model_lazy</code> object.
.method	A string naming the method variant. Must match a named variant() in the agendas() of the matched stat_define() . E.g. "boot", "permute", "permute_rfast".
...	Named arguments forwarded to the variant.

Value

The modified lazy object with `recalibrate_spec` populated.

See Also

[conclude\(\)](#), [stat_define\(\)](#)

Examples

```
sleep |>
  define_model(x_by(extra, group)) |>
  prepare_test(T_TEST) |>
  via("boot", n = 2000) |>
  conclude()
```

```
sleep |>
  define_model(x_by(extra, group)) |>
  prepare_test(T_TEST) |>
  via("permute", n = 999L) |>
  conclude()
```

write_models

Write multiple model definitions from a data frame

Description

`write_models()` evaluates named model expressions sequentially against `.data`, so each name is available to subsequent expressions via `stats::update()`. Accepts any valid variable mapper `<var_id>`: `<formulas>`, [rel\(\)](#), [x_by\(\)](#), or any registered `var_id` type.

Usage

```
write_models(.data, ...)
```

Arguments

<code>.data</code>	A data frame.
<code>...</code>	Named model expressions. Each must evaluate to a formula or a <code>var_id</code> object. Names are used as row labels in anova() output and as the <code>model</code> column in tidy() .

Details

Sits between a data frame and [prepare_model\(\)](#) or [prepare_test\(\)](#) in the pipeline.

Value

An `expanded_model` object.

See Also

[prepare_model\(\)](#), [prepare_test\(\)](#), [anova\(\)](#), [conclude\(\)](#), [display\(\)](#)

Examples

```

# explicit formulas
LifeCycleSavings |>
  write_models(
    f1 = sr ~ 1,
    f2 = sr ~ pop15,
    f3 = sr ~ pop15 + pop75,
    f4 = sr ~ pop15 + pop75 + dpi,
    f5 = sr ~ pop15 + pop75 + dpi + ddpi
  ) |>
  prepare_model(LINEAR_REG) |>
  anova()

# update() chain (formulas only)
LifeCycleSavings |>
  write_models(
    f1 = sr ~ 1,
    f2 = update(f1, ~. + pop15),
    f3 = update(f2, ~. + pop75),
    f4 = update(f3, ~. + dpi),
    f5 = update(f4, ~. + ddpi)
  ) |>
  prepare_model(LINEAR_REG) |>
  anova()

# conclude() -> returns a multi_exec
LifeCycleSavings |>
  write_models(
    f1 = sr ~ 1,
    f2 = sr ~ pop15,
    f3 = sr ~ pop15 + pop75
  ) |>
  prepare_model(LINEAR_REG) |>
  conclude()

## Not run:
# display() -> show up to n models in full
LifeCycleSavings |>
  write_models(
    f1 = sr ~ 1,
    f2 = sr ~ pop15,
    f3 = sr ~ pop15 + pop75,
    f4 = sr ~ pop15 + pop75 + dpi,
    f5 = sr ~ pop15 + pop75 + dpi + ddpi
  ) |>
  prepare_model(LINEAR_REG) |>
  conclude() |>
  display(5)

## End(Not run)

# via rel()

```

```
mtcars |>
  define_model(rel(wt, mpg)) |>
  prepare_model(LINEAR_REG) |>
  anova()
mtcars |>
  write_models(
    m1 = rel(wt, mpg),
    m2 = rel(hp, mpg)
  ) |>
  prepare_model(LINEAR_REG) |>
  anova()

# mixed var_id types in a single write_models() call
suppressWarnings({
  mtcars |>
    write_models(
      null = mpg ~ 1,
      m1 = rel(wt, mpg),
      m2 = rel(hp, mpg)
    ) |>
    prepare_model(LINEAR_REG) |>
    conclude()
})

# via prepare_test()
mtcars |>
  write_models(
    by_am = x_by(mpg, am),
    by_vs = x_by(mpg, vs)
  ) |>
  prepare_test(T_TEST) |>
  conclude()
```

x_by

Compare a variable by group

Description

x_by() (and its infix alias %by%) creates an x_by Variable Mapper that reads as "compare x by group". Expressions are captured unevaluated, similar to how `ggplot2::aes()` captures aesthetics.

Usage

```
x_by(x, group)
```

```
x %by% group
```

Arguments

x	The response variable. Accepts a bare name, a <code>c()</code> of bare names, a <code>tidyselect</code> helper (requires data in <code>define_model()</code>), or <code>I(expr)</code> for inline data.
group	The grouping variable. Same rules as x.

Value

An `x_by / var_id` S7 object.

Examples

```
# Bare names – resolved later from the data or environment
x_by(extra, group)

# Infix alias: identical to x_by(extra, group)
extra %by% group

# Inline data via I()
x_by(I(rnorm(30)), I(rep(c("a", "b"), each = 15)))

# Named inline data
x_by(I(score = rnorm(30)), I(grp = rep(c("a", "b"), each = 15)))
```

Index

- * **datasets**
 - anova-mod, 8
 - predict, 50
- %<-% (modifying-assignment), 43
- %by% (x_by), 66

- add-stat-define, 4
- add-variant, 6
- add_stat_define (add-stat-define), 4
- add_variant (add-variant), 6
- agendas, 7
- agendas(), 4, 5, 7, 13, 41, 56, 57, 63
- anova (anova-mod), 8
- anova(), 17, 19, 20, 36, 64
- anova-mod, 8
- anova_able, 17–19
- auto_gauge, 9
- auto_gauge(), 32, 38, 40
- auto_predict, 10
- auto_predict(), 38, 41, 51
- auto_tidy, 11
- auto_tidy(), 12, 13, 16, 17, 21–26, 28, 29, 39, 42, 58, 60, 62, 63

- baseline, 12
- baseline(), 7, 8, 11, 22, 23, 27, 28, 40, 56, 57, 63

- check_param_nodes
 - (claim-vars-validators), 13
- check_x_and_given
 - (claim-vars-validators), 13
- check_x_and_given(), 14
- claim-vars-validators, 13
- claim_contrast_coefs, 15
- claim_contrast_coefs(), 16
- claim_scalar, 16
- class_corr_two, 16, 29
- class_glm_object, 17
- class_lm_object, 19, 22, 23, 36

- class_p_test, 21, 47, 48
- class_stat_infer, 10–13, 16, 17, 21, 22, 22, 24–26, 28, 38–42, 60, 62, 63
- class_ttest_one, 24
- class_ttest_pairwise, 25, 58
- class_ttest_two, 26, 58
- class_var_inform, 27
- conclude, 27
- conclude(), 8, 29–32, 34, 43, 48, 51–53, 57–60, 64
- COR_TEST, 16, 17, 29
- cortest-formula, 29
- cortest-rel, 29

- define_model (layout-define-base), 35
- define_model(), 13, 27, 32, 36, 46, 49, 51–53, 55, 67
- display, 30
- display(), 64

- gauge, 31
- gauge(), 10
- ggplot2::aes(), 45, 55, 66
- GLM, 17, 18, 32

- HTEST_FN, 33
- HTEST_FN(), 4, 6, 37, 39, 43, 56, 57

- I(), 34
- inlines, 34

- layout-define-base, 35
- LINEAR_REG, 19, 20, 36, 51
- LINEAR_REG(), 52

- making_gauge, 37
- making_gauge(), 10, 32, 40
- making_predict, 38
- making_predict(), 11, 41, 51
- making_tidy, 39
- making_tidy(), 11, 12, 22, 23, 28, 41–43, 60

- map_claim, 39
- map_claim(), 12, 13, 63
- method_gauge, 40
- method_gauge(), 10, 38
- method_predict, 41
- method_predict(), 11, 38
- method_tidy, 41
- method_tidy(), 12, 39, 43, 60
- model-processor, 42
- MODEL_FN, 43
- MODEL_FN(), 4, 6, 34, 37–39, 56, 57
- model_infer_define
 - (stat-infer-definer), 56
- model_processor (model-processor), 42
- model_processor(), 7, 12, 13, 27, 28, 56, 57, 61–63
- modifying-assignment, 43
- MU, 44
- MU(), 15, 16
- null-hyp, 45
- on, 45
- on(), 24
- P_TEST, 21, 22, 46
- P_TEST(), 47
- pairwise, 48
- pairwise(), 25, 34, 35, 46, 60, 61
- param_obj, 16, 49
- param_obj(), 15
- PI, 50
- PI(), 15, 16, 47, 48
- predict, 50
- predict(), 11, 18, 20
- prepare, 51
- prepare-model, 52
- prepare-test, 53
- prepare_model (prepare-model), 52
- prepare_model(), 8, 27, 28, 32, 36, 43, 51, 52, 64
- prepare_test (prepare-test), 53
- prepare_test(), 27, 28, 34, 35, 45, 51, 52, 57, 64
- print(), 17, 22, 24–26
- prop, 54
- prop(), 21, 42, 46–48, 61
- purge_stat_defines, 54
- purge_stat_defines(), 4, 5
- rel, 55
- rel(), 16, 17, 29, 34–36, 46, 60, 61, 64
- remove_stat_define (add-stat-define), 4
- remove_stat_define(), 5
- remove_variant (add-variant), 6
- RHO, 55
- RHO(), 14–16
- stat-infer-definer, 56
- STAT_CONSTRUCTOR, 57
- STAT_CONSTRUCTOR(), 51
- stat_define (stat-infer-definer), 56
- stat_define(), 4, 5, 7, 8, 13, 33, 34, 43, 63, 64
- state_null (null-hyp), 45
- state_null(), 12, 13, 15, 24, 29, 40, 47, 48, 58, 63
- stats::binom.test(), 47, 48
- stats::binomial(), 32
- stats::gaussian(), 32
- stats::glm(), 32
- stats::poisson(), 32
- stats::prop.test(), 47, 48
- stats::t.test(), 25
- stats::update(), 64
- T_TEST, 24–26, 51, 53, 57
- T_TEST(), 33
- test_define (stat-infer-definer), 56
- test_define(), 57
- tidy, 59
- tidy(), 11, 12, 23, 31, 60, 64
- ttest-formula, 58
- ttest-on, 24, 58
- ttest-pairwise, 25, 58
- ttest-xby, 58
- validate_claim_vars
 - (claim-vars-validators), 13
- validate_one_param_node
 - (claim-vars-validators), 13
- var_id, 13, 14, 60
- var_id(), 15, 49
- var_id_info, 61
- var_id_info(), 27
- variant, 62
- variant(), 6–8, 11, 13, 22, 23, 27, 28, 40, 56, 57, 63
- via, 63

`via()`, [27–29](#), [34](#), [43](#), [47](#), [48](#), [51–53](#), [57](#), [58](#),
[62](#), [63](#)

`write_models`, [64](#)

`write_models()`, [8](#), [30](#), [31](#), [51–53](#)

`x_by`, [66](#)

`x_by()`, [26](#), [34](#), [35](#), [42](#), [46](#), [60](#), [61](#), [64](#)