Package: jstats (via r-universe)

October 27, 2026

Title Simplified Statistical Analysis Tools for Social Science

Version 0.9.267

Description Provides simplified tools for common statistical analyses used in social science research and teaching, with output formatted in the style of 'SPSS', 'Stata', or 'SAS'. Analysis functions include descriptive statistics, frequency tables, t-tests, ANOVA, correlations, chi-square tests, cross-tabulations, linear regression, logistic regression, and Cronbach's alpha, with built-in model diagnostics and plotting. Supporting tools handle data management (recoding, labelling, encoding text as coded values, case selection, dummy coding, scale construction, data screening) and unified data import and export for 'SPSS', 'Stata', 'SAS', 'Excel', CSV, and R formats. Declared missing values are read and written faithfully in each platform's own convention, can be declared or converted between conventions within R, and suspected but undeclared missing-value codes are detected automatically. Functions accept unquoted variable names and formula syntax, handle 'haven'-labelled variables directly, and return results invisibly while printing readable tables. The package stays close enough to base R conventions that users learn transferable skills rather than a private dialect, easing the transition from 'SPSS', 'Stata', or 'SAS' into the broader R ecosystem.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

Depends R (>= 4.2.0)

Roxygen list(markdown = TRUE)

Imports ggplot2, haven (>= 2.3.0), labelled, rlang, readxl, writexl, callr, tools

Suggests generics

LazyData true

URL https://JMA61.github.io/jstats/

BugReports https://github.com/JMA61/jstats/issues

Config/roxygen2/version 8.0.0

Config/pak/sysreqs make libicu-dev libx11-dev zlib1g-dev

Repository https://jma61.r-universe.dev

Date/Publication 2026-10-27 10:32:22 UTC

RemoteUrl https://github.com/JMA61/jstats

RemoteRef 0.9.267

RemoteSha 32edb70bbc6b8aa01f49d26cb492d98e376e5542

Contents

clinic . . . 3
community . . . 4
jai . . . 6
jalpha . . . 7
jaov . . . 9
javg . . . 13
jcomplete . . . 15
jconvert . . . 17
jcopy . . . 21
jcorr . . . 22
jcount . . . 25
jcrosstab . . . 26
jdata_dir . . . 29
jdeclare_missing . . . 30
jdesc . . . 37
jdummy . . . 40
jencode . . . 42
jfreq . . . 46
jlikert . . . 49
jlm . . . 50
jload . . . 58
jlogistic . . . 65
jnumeric . . . 71
joptions . . . 72
joutput . . . 76
jplot . . . 82
jrecode . . . 87
jrelabel . . . 93
jsave . . . 97
jscreen . . . 100
jsubset . . . 104
jsum . . . 107
jt . . . 108

jupdate 112
juse 114

Index **115**

`clinic` *Clinic intervention example dataset*

Description

A small synthetic mental-health and wellbeing intervention sample, used as the package's messy-data companion to `community`. Where `community` is the clean default, `clinic` deliberately carries undeclared missing-value codes, a column whose value labels were stripped on import, and an imperfect scale item, so the declare-and-clean workflow has realistic material to work on. The 70 clients and 16 variables echo the teaching structure of `community` – an interaction, a null variable, non-overlapping missingness, a recode dichotomy, a clean logistic outcome, a multi-category variable, and a Likert battery – in a psychology setting. The data are synthetic, but the relationships among the variables are realistic.

Usage

```
clinic
```

Format

A data frame with 70 rows and 16 variables:

ClientID Client ID, character ("C001", "C002", ...).

Stress Perceived stress (integer, 0-40). Carries SPSS-style missing values (-99 Refused, -98 Don't know).

SocialSupport Perceived social support (integer, 0-24); the buffering partner in the Stress-by-SocialSupport interaction on Flourishing.

SleepHours Average nightly sleep in hours. Carries SPSS-style missing values (-99 Refused, -98 Don't know), placed on cases that do not overlap the Stress codes, so a model using both predictors drops more cases than either alone.

Flourishing Flourishing score (integer, 0-100); built with a Stress-by-SocialSupport interaction (the buffering hypothesis).

ScreenTime Daily screen time in hours; deliberately near-independent of the other variables.

PriorTherapy Received therapy before the study, dichotomy coded 1/2 (1 Yes, 2 No); recode to 0/1 before use as a logistic-regression outcome.

SoughtHelp Sought professional help during the study, dichotomy coded 0/1 (0 No, 1 Yes). A clean logistic-regression outcome.

Medication Currently taking medication, dichotomy coded 0/1 (0 No, 1 Yes). Carries an SPSS-style missing value (-99 Refused).

Condition Treatment condition, 4 categories (1 Control, 2 CBT, 3 Mindfulness, 4 Support group); has a modest effect on Flourishing.

MoodRating Mood rating (integer, 1-10). Arrives "dirty": literal -99 (Refused) and -98 (Don't know) codes are present in the data with NO missing-value declaration, the state of play after a CSV or Excel import. The package's jdeclare_missing() demonstration variable: summary statistics are poisoned until the codes are declared.

Anxiety1 "I felt calm and relaxed." 5-point Likert (1 Not at all to 5 Extremely); reverse-keyed (the variable label ends in " R"). Reverse-code before scale scoring.

Anxiety2 "I worried about many different things." 5-point Likert. Arrives with literal -99/-98 codes present in the data and NO missing-value declaration – the undeclared contrast to Anxiety4.

Anxiety3 "I felt afraid for no clear reason." 5-point Likert; arrives with its value labels stripped (a plain numeric column, as after a CSV import that dropped the labels).

Anxiety4 "I had trouble controlling my worry." 5-point Likert. Carries properly declared SPSS-style missing values (-99 Refused, -98 Don't know) – the declared contrast to Anxiety2.

Anxiety5 "I felt restless or on edge." 5-point Likert; weakly loaded (a Cronbach's-alpha drop candidate when scoring the scale).

Details

The five Anxiety items form a single scale, with one deliberate problem per item: Anxiety1 is reverse-keyed; Anxiety2 carries literal -99/-98 codes that are not declared as missing; Anxiety3 arrives with its value labels stripped; Anxiety4 carries the same -99/-98 codes but properly declared (the clean contrast to Anxiety2); and Anxiety5 is the weak item that scale-reliability output flags for dropping. Stress and SleepHours carry SPSS-style missing values on non-overlapping cases, so listwise deletion across both reduces the analysis sample below the per-variable counts. MoodRating and Anxiety2 are the two columns whose -99/-98 codes arrive undeclared, awaiting `jdeclare_missing()`. The Stress-by-SocialSupport interaction on Flourishing is the buffering hypothesis (higher social support weakens the negative association between stress and flourishing), and treatment Condition has a modest effect on Flourishing.

Source

Synthetic data generated by `data-raw/clinic_data_generator.R` (random seed 20260614).

See Also

`community`, the clean default example dataset.

`community` *Community survey example dataset*

Description

A small synthetic survey dataset used throughout the package as a runnable example. It backs the function examples, serves as a teaching dataset for new users, and demonstrates cross-platform save and load behavior. The 100 respondents and 15 variables are chosen to exercise the kinds of data social-science users actually have: Likert scales, dichotomies, a multi-category variable, continuous measures, and SPSS-style missing values. The data are synthetic, but the relationships among the variables are realistic.

Usage

```
community
```

Format

A data frame with 103 rows and 15 variables:

RespondentID Respondent ID, character ("R001", "R002", ...).

Income Annual income (USD). Carries SPSS-style missing values (-99 Refused, -98 Don't know).

Education Highest education level, 5 categories (1 Some high school, 2 High school graduate, 3 Some college, 4 Bachelor's degree, 5 Graduate degree). Carries SPSS-style missing values (-99 Refused, -98 Don't know).

Age Age in years (integer, 18-80).

WellbeingScore Flourishing score (integer, 0-100); built with an Income-by-Age interaction.

Volunteer Volunteered in past year, dichotomy coded 0/1 (0 No, 1 Yes).

OwnsHome Owns home, dichotomy coded 1/2 (1 Yes, 2 No); recode to 0/1 before use as a logistic-regression outcome.

Smoker Current smoker, dichotomy coded 0/1 (0 No, 1 Yes). Carries an SPSS-style missing value (-99 Refused).

CommuteTime Daily commute time in minutes (integer); deliberately near-independent of the other variables.

Region Region of residence, 4 categories (1 North, 2 South, 3 East, 4 West).

Environment1 "Climate change is a serious threat." 5-point Likert (1 Strongly Disagree to 5 Strongly Agree). Carries SPSS-style missing values (-99 Refused, -98 Don't know).

Environment2 "Concern about the environment is exaggerated." 5-point Likert; reverse-keyed (the variable label ends in " R"). Reverse-code before scale scoring.

Environment3 "Government should do more for the environment." 5-point Likert. Carries SPSS-style missing values (-99 Refused, -98 Don't know).

Environment4 "I would pay more for environmentally friendly products." 5-point Likert.

Environment5 "Pollution is a major cause of public health problems." 5-point Likert; weakly loaded (a Cronbach's-alpha drop candidate when scoring the scale).

Details

`community` is the clean default example dataset. For a companion dataset that deliberately carries undeclared missing-value codes, stripped value labels, and an imperfect scale – the material the data-cleaning workflow operates on – see `clinic`.

The five Environment items form a single attitude scale: item 2 is reverse-keyed, and item 5 is the weak item that scale-reliability output flags for dropping. Income, Education, Smoker, Environment1, and Environment3 carry SPSS-style missing values, with the codes placed on partly non-overlapping cases so that listwise deletion visibly reduces the analysis sample below the per-variable counts. All of community's missing-value codes are properly declared; for a dataset with undeclared codes and other deliberate data-cleaning problems, see `clinic`.

Source

Synthetic data generated by `data-raw/community_data_generator.R` (random seed 20260605).

See Also

`clinic`, the messy-data companion dataset.

`jai` *Print the jstats orientation, or install it for an AI assistant*

Description

`jai()` prints a short, plain-text orientation to the package's core conventions: how to load data, how jstats handles value labels and declared missing values, how to choose an analysis function, and how to keep changes made to a data frame. It is written to be useful both to people new to the package and to AI coding assistants (such as the assistant built into RStudio), which read console output and can act on what they find there.

Usage

```
jai(setup = NULL, path = NULL, ...)
```

Arguments

`setup`	Optional. `"project"`, `"machine"`, `"chat"`, or `"status"`; see Details. When missing, the orientation is printed to the console.
`path`	Optional. An existing folder to write into, overriding the default destination; used only by `"project"` and `"machine"`.
`...`	Not used. An input `jai()` does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Details

Beyond the plain printout, `jai()` can install the same orientation where an AI assistant finds it on its own. `setup` selects the situation (values are case-insensitive):

`"project"` Writes the orientation into `AGENTS.md` in the current folder (or `path`), inside a clearly marked block. Assistants that read `AGENTS.md` then see the conventions in every conversation in that project. Existing content is never overwritten: the block is appended to an existing file, and on regeneration only the marked block is replaced. Keep your own additions outside the markers; they survive regeneration, while edits inside the block are overwritten (with a warning when edits are detected).

`"machine"` Writes `SKILL.md` to the user-level skills folder (or `path`), so assistants that support skills can load the conventions in any project on the machine, when relevant.

`"chat"` For chat assistants outside RStudio. Currently prints a short note; a paste-ready primer is planned.

`"status"` Reports which orientation files are present, their versions, and whether they are current. Nothing is written.

The file-writing situations confirm the exact destination before writing (or write without asking when `path` names the folder yourself). In a session that cannot ask – a script run with Rscript, a knitr render – they stop before the prompt and say so; name the folder with `path =` to write there, as `jsave()` and `jload()` do with their own overwrite prompts. Each written file carries a version stamp; after updating jstats, rerun the same `jai()` call to refresh it.

Value

Invisibly `NULL`. Called for its side effects.

See Also

`help("jstats")` for the package overview and full function list.

Examples

```
jai()
## Not run:
jai("project")   # write AGENTS.md in the current project
jai("machine")   # write SKILL.md to the skills folder
jai("status")    # report what is installed where

## End(Not run)
```

---

jalpha *Cronbach's Alpha Reliability Analysis*

Description

Computes Cronbach's alpha and prints SPSS-style reliability output including a case processing summary, overall alpha, item statistics, and item-total statistics with alpha-if-item-deleted. Built from scratch with no external package dependencies beyond base R. Handles haven-labelled variables automatically. Detects potentially reverse-coded or misfit items.

Usage

```
jalpha(
  data,
  ...,
  subset = NULL,
  variable.id = NULL,
  value.id = NULL,
  case.processing.detail = NULL,
  digits = NULL,
  print = NULL
)
```

Arguments

data	A data frame.
...	Unquoted variable names (scale items) within data. Use colon notation (e.g. Item1:Item6) to select a range of consecutive columns.
subset	An optional unquoted logical expression (e.g. Group == 1) to subset cases for this call only. Applied after jcomplete and jsubset. Does not affect other function calls. Written in R syntax and checked the way jsubset() checks a filter: subset = NOT(Age < 40), for example, is refused with the corrected subset = !(Age < 40) shown (see jsubset for a translation table).
variable.id	Character or NULL. Variable label display mode: one of "both", "names", "labels", "legend", or "legend.bottom". "names" shows variable names only; "both" shows "name: label"; "labels" shows each item's label in the Item column of the Item Statistics and Item-Total Statistics tables (best for short labels; the returned tables and the reverse-coding diagnostic keep variable names); "legend"/"legend.bottom" keep names and print a label legend after the final table. NULL (default) defers to joutput()'s variable.id setting. Not a logical.
value.id	Not supported by jalpha(). The function does not display value labels, so passing this argument is an error. It exists only to return a clear message rather than report value.id as a variable that was not found. Leave at NULL (default).
case.processing.detail	Per-call override of the Case Processing Summary detail tier: one of "none", "totals", or "per_code". NULL (default) uses the active joutput() level default. The Case Processing table itself prints only when a filter or listwise deletion excluded cases; otherwise a one-line N statement takes its place. See ?joutput (case.processing).
digits	Integer or NULL. Number of decimal places for continuous statistics in the output tables (range 0-7; digits = 0 prints whole numbers with no trailing decimal point). Does not affect p-values, percentages, or integer quantities (counts, N, degrees of freedom), which keep their own fixed conventions. NULL (default) defers to joutput()'s digits setting (default 3).
print	Logical or NULL. FALSE prints nothing – no table, no title, no note – and returns the result as every call does; errors and warnings still show. TRUE prints even under joutput("silent"). NULL (the default) prints unless the level is silent. For a script that keeps many results and wants no output: see joutput.

Details

A red "Reliability Analysis" title is printed first, followed by the case processing summary, overall alpha, item statistics, and item-total statistics.

Value

Invisibly returns a list of class jst_alpha containing: alpha (Cronbach's alpha), n_items, n_used, n_excluded, item_statistics, item_total_statistics, and sample_info (pipeline and missing data counts). item statistics data frame, and item-total statistics data frame.

See Also

jstats for the package overview, workflow conventions, and complete function listing.

Examples

```
# With explicit data frame
jalpha(community, Environment1, Environment2, Environment3,
       Environment4, Environment5)

# The data frame may also be given by name
jalpha(data = community, Environment1, Environment2, Environment3,
       Environment4, Environment5)

# Using juse() default
juse(community)
jalpha(Environment1, Environment2, Environment3, Environment4,
       Environment5)
```

---

jaov *One-way ANOVA (traditional or Welch method)*

Description

Runs a one-way ANOVA and prints a formatted group descriptives table followed by an ANOVA table. By default, runs the traditional ANOVA assuming equal variances. Optional parameters provide post-hoc tests, effect size, Levene's test, and confidence intervals. Set welch = TRUE for the Welch correction when equal variances cannot be assumed; its post-hoc test is Games-Howell. Handles haven-labelled, numeric, and factor grouping variables. For haven-labelled variables, numeric codes are displayed alongside labels in the group descriptives table.

Usage

```
jaov(
  formula,
  data,
  welch = FALSE,
  posthoc = NULL,
  effect.size = NULL,
  diagnostics = NULL,
  ci = NULL,
  subset = NULL,
  variable.id = NULL,
  value.id = NULL,
  case.processing.detail = NULL,
  full = FALSE,
  digits = NULL,
  print = NULL,
```

```
  ...
)
```

Arguments

formula	A formula of the form `DV ~ Group`. A transformed term such as `log(DV)` is computed automatically: the tests and the descriptive output all use the transformed values.
data	A data frame containing variables referenced in `formula`.
welch	Logical. If FALSE (default), runs traditional ANOVA. If TRUE, runs Welch's ANOVA (does not assume equal variances). Welch's F is not a ratio of two mean squares, so its table shows F, df1, df2 and p: Sum of Squares and Mean Square belong to the traditional ANOVA table and are not applicable to Welch's test. Welch's ANOVA needs at least 2 cases in every group, and some variation within every group: a group whose cases all hold one value has a variance of zero, which Welch's F divides by. The traditional ANOVA can include such a group, or a group of one case.
posthoc	Logical or NULL. If TRUE, prints pairwise comparisons of the group means: Tukey HSD for the traditional ANOVA, and Games-Howell when welch = TRUE. Games-Howell does not assume equal variances: each comparison uses the two groups' own variances and its own degrees of freedom, shown in a df column, and the p-values and confidence intervals are adjusted for the number of groups through the studentized range distribution, as Tukey's are. Commercial statistical software offers the same test for unequal variances (Games-Howell in SPSS's ONEWAY post-hoc tests). A Games-Howell comparison with fewer than 2 degrees of freedom – possible when a group has two or three cases – shows its difference and its df and leaves its interval and p-value blank, with a line under the table saying so: the studentized range distribution is not defined there. With two groups no post-hoc table is printed or returned, after either ANOVA: the test above it is the only comparison. If NULL (default), defers to `joutput()`.
effect.size	Logical or NULL. If TRUE, prints eta-squared. If NULL (default), defers to `joutput()`.
diagnostics	Logical, character vector, or NULL. If TRUE, prints Levene's test for homogeneity of variance, with a note under it when the test is significant (see "Unequal variances"), and draws a normal Q-Q plot of the standardized residuals – each case's distance from its group's mean – a look at normality as a plot rather than a test. `"levene"` or `"qq"` asks for one of them; `c("levene", "qq")` for both. If NULL (default), defers to `joutput()`'s `diagnostics` setting, which is off at every output level until it is set. The Q-Q plot is also drawn by `jplot(result, which = "qq")`.
ci	Logical or NULL. If TRUE, adds 95% confidence intervals to the group descriptives table. If NULL (default), defers to `joutput()`.
subset	An optional unquoted logical expression (e.g. `Group == 1`) to subset cases for this call only. Applied after jcomplete and jsubset. Does not affect other function calls. Written in R syntax and checked the way `jsubset()` checks a filter: `subset = NOT(Age < 40)`, for example, is refused with the corrected `subset = !(Age < 40)` shown (see `jsubset` for a translation table).

variable.id — Character or NULL. Variable label display mode: one of "both", "names", "labels", "legend", or "legend.bottom". "names" shows variable names only; "both" shows "name: label"; "labels" shows the DV and grouping-variable labels wherever the variable name appears (table captions and the ANOVA Source row; group levels follow the value.id mode) – best for short labels; "legend"/"legend.bottom" keep names and print a label legend after the output. NULL (default) defers to joutput()'s variable.id setting. Not a logical.

value.id — Character or NULL. Value-label display mode for the group descriptives rows: "both" ("code: label"), "values" (bare code), or "labels" (the label, degrading to the bare code where a code has none). "legend" and "legend.bottom" keep the bare code in the table and print a value-label legend after it ("legend" per-table, "legend.bottom" consolidated where multiple tables are produced). A no-op for grouping variables with no value labels. NULL (default) defers to joutput()'s value.id setting. Not a logical.

case.processing.detail — Per-call override of the Case Processing Summary detail tier: one of "none", "totals", or "per_code". NULL (default) uses the active joutput() level default. The Case Processing table itself prints only when a filter or listwise deletion excluded cases; otherwise a one-line N statement takes its place. See ?joutput (case.processing).

full — Logical. If TRUE, turns on posthoc, effect.size and ci together. Does not override explicit FALSE values, and does not turn on diagnostics, which diagnostics alone governs.

digits — Integer or NULL. Number of decimal places for continuous statistics in the output tables (range 0-7; digits = 0 prints whole numbers with no trailing decimal point). Does not affect p-values, percentages, or integer quantities (counts, N, degrees of freedom), which keep their own fixed conventions. NULL (default) defers to joutput()'s digits setting (default 3).

print — Logical or NULL. FALSE prints nothing – no table, no title, no note – and returns the result as every call does; errors and warnings still show. TRUE prints even under joutput("silent"). NULL (the default) prints unless the level is silent. For a script that keeps many results and wants no output: see joutput.

... — Reserved for argument-name checking. Passing levene, the name of the diagnostics setting before version 0.9.219, produces an error that names diagnostics.

Details

A red title identifying the test type is printed first, followed by variable labels (if present), then the results tables.

A transformed outcome or grouping term in formula – log(x) and the like – is computed once on the analysis data and used by the F test, Levene's test, the post hoc comparisons, and the descriptives, so they all describe the same values. The transforms supported inline, and those that must be created as a column first, are as documented for jlm. A value or vector from your workspace may be named inside a computed term, as in lm(): I(x > cutoff) with cutoff <- 10. A data frame may not: write y ~ x with data = MyData, not MyData$y ~ MyData$x. A variable may not: jstats analyzes the variables of a data frame, so a vector holding one value for each case – w in I(x * w), or an object reached with $ such as lst$w – is added to the data frame first (MyData$w <- w), where

`lm()` would read it from the workspace; the stop gives that line. A vector of any other length used value by value is refused, where `lm()` would repeat the shorter one; a set used with `%in%` may have any length.

Value

Invisibly returns a list of class `jst_anova` containing: `model` (the `aov` or `oneway.test` object), `model_frame` (the analysis data frame used for plotting), `test_type`, `formula`, `descriptives`, `f`, `df1`, `df2`, `p`, `eta_squared`, `n`, `sample_info` (pipeline and missing data counts), and `posthoc` (the pairwise comparisons as an unrounded data frame, with `test` naming the method; `NULL` when post-hoc tests were not requested), and `levene`: Levene's test as printed, a list of `F`, `df1`, `df2`, `p` and `note` – the verdict of the note under a significant test (`"acceptable"`, `"unreliable"` or `"differ"`), recorded at every output level, or `NA` when no note applies (the test not significant, or Welch's ANOVA run); `NULL` when Levene's test was not asked for.

Unequal variances

The traditional ANOVA assumes the groups have the same variance, and Levene's test (`diagnostics = TRUE`) asks whether they do. When it is significant, a note under its table states the two things that decide how much that matters – the ratio of the largest group's size to the smallest, and of the largest standard deviation to the smallest – and then takes one of three forms. Textbooks give different guidelines and no single cutoff is agreed, so the note does not treat one as exact:

- Stevens (*Applied Multivariate Statistics for the Social Sciences*; *Intermediate Statistics: A Modern Approach*) treats the F test as robust to unequal variances while the largest group is less than 1.5 times the smallest.
- Moore, McCabe and Craig (*Introduction to the Practice of Statistics*) treat the results as approximately correct while the largest standard deviation is less than twice the smallest; Howell (*Statistical Methods for Psychology*) gives the same limit as a variance ratio of four and adds that unequal variances and unequal group sizes do not mix.

The note reads "usually still acceptable" when the largest group is no more than 1.25 times the smallest and the largest standard deviation no more than twice the smallest; it says the p-value may not be reliable when the group sizes differ by more than 1.5 times and the standard deviations by more than twice; and between the two it says that guidelines differ. The 1.25 is not a textbook figure. It comes from simulations run for jstats (a true null hypothesis, normal scores, a nominal 5 percent level, four groups, the smallest group the most variable): in the cases tried, the traditional ANOVA rejected up to about 7.5 percent of the time inside the first range, about 7 to 13 percent in the middle one, and 13 to 16 percent in the last. The direction matters: those figures are for the worst case, and when the LARGEST group is the most variable the rate is lower, falling under 5 percent where the sizes are clearly unequal. Welch's ANOVA (`welch = TRUE`) does not assume equal variances. The note is not printed at `joutput("minimal")`, which prints the table alone.

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
# With explicit data frame
jaov(WellbeingScore ~ Region, data = community)
jaov(WellbeingScore ~ Region, data = community, full = TRUE)

# The data frame may also be given positionally
jaov(WellbeingScore ~ Region, community)

# Checking the equal-variances assumption: Levene's test
jaov(WellbeingScore ~ Region, data = community, diagnostics = TRUE)

# Pairwise comparisons after the traditional ANOVA: Tukey HSD
jaov(WellbeingScore ~ Region, data = community, posthoc = TRUE)

# When equal variances cannot be assumed: Welch's ANOVA
jaov(WellbeingScore ~ Region, data = community, welch = TRUE)

# Pairwise comparisons after Welch's ANOVA: Games-Howell
jaov(WellbeingScore ~ Region, data = community, welch = TRUE,
     posthoc = TRUE)

# Using juse() default
juse(community)
jaov(WellbeingScore ~ Region)
jaov(WellbeingScore ~ Region, full = TRUE)
```

---

javg *Compute a row-wise mean across multiple variables*

Description

`javg()` computes the mean of values across multiple variables for each case (row) in the data frame. This is typically used to create scale means from a set of related items.

By default, cases with any missing values receive `NA`. Use the `min.valid` argument to allow partial means — for example, `min.valid = 1` computes the mean of available values as long as at least one item is non-missing.

By default, the denominator is the number of non-missing values for each case. Use `fixed = TRUE` to always divide by the total number of variables regardless of missing values.

Variables can be listed individually or using colon notation to select a range of consecutive columns (e.g. `Attitude1:Attitude6`).

Usage

```
javg(data, ..., min.valid = NULL, fixed = FALSE, var.label = NULL)
```

Arguments

data	A data frame, or omit to use the juse() default.
...	Unquoted variable names. Use colon notation (e.g. Attitude1:Attitude6) to select a range of consecutive columns.
min.valid	Integer (optional). The minimum number of non-missing values required to compute a mean. If a case has fewer non-missing values, the result is NA. If omitted, all values must be non-missing (equivalent to setting min.valid to the number of variables).
fixed	Logical. If FALSE (default), the denominator for each case is the number of non-missing values (i.e. the mean adjusts for missing data). If TRUE, the denominator is always the total number of variables (i.e. missing values effectively count as zero).
var.label	Character string (optional). A variable label to attach to the result. If omitted, an auto-generated label is used.

Value

A numeric vector the same length as nrow(data), suitable for assigning to a new column: MyData$ScaleMean <- javg(Var1, Var2, Var3).

See Also

jsum for computing row-wise sums.

jstats for the package overview, workflow conventions, and complete function listing.

Examples

```
# Set the default data frame (so you can omit it in function calls)
juse(community)

# Mean of three variables (all must be non-missing)
community$EnvAvg <- javg(Environment1, Environment3, Environment4)

# Mean with partial data allowed (at least 2 non-missing)
community$EnvAvg <- javg(Environment1, Environment3, Environment4,
                         min.valid = 2)

# Mean using colon range for consecutive columns
community$ScaleMean <- javg(Environment1:Environment5)

# Mix colon ranges and explicit names (e.g. after reverse-coding an item)
community$Environment2R <- jrecode(community, Environment2,
                                   map = "1=5; 2=4; 3=3; 4=2; 5=1")
community$ScaleMean <- javg(Environment1, Environment2R,
                            Environment3:Environment5)

# Fixed denominator (always divide by total number of variables)
community$EnvAvg <- javg(Environment1, Environment3, Environment4,
                         min.valid = 2, fixed = TRUE)
```

```
# With a custom variable label
community$ScaleMean <- javg(Environment1:Environment5,
                            var.label = "Environment Scale Mean")

# With an explicit data frame (instead of using juse default),
# positional or named
community$EnvAvg <- javg(community, Environment1, Environment3,
                         Environment4)
community$EnvAvg <- javg(data = community, Environment1, Environment3,
                         Environment4)

# juse(NULL) clears the default dataset -- not normally needed. You'd
# clear a default or registration only to undo a mistake, or -- as in
# this example -- to reset state for testing.
juse(NULL)
```

`jcomplete` *Set a listwise complete-case filter for matching N across analyses*

Description

`jcomplete()` registers a set of variables and activates a listwise deletion filter that excludes any case with a missing value on any of the registered variables. This ensures that all subsequent analyses use the same set of complete cases, which is essential when preliminary analyses need to match the N of a final regression model.

The setting is stored per dataset, so switching `juse()` between datasets preserves each dataset's setting independently.

The jcomplete filter applies whenever the matching dataset is used, regardless of whether it was supplied via `juse()` or specified explicitly in a function call. To bypass temporarily without losing the setting, use `jcomplete(off)` before the analysis and `jcomplete(on)` afterward. This matches the SPSS USE ALL / FILTER convention.

If a variable in the setting is later dropped from the dataset or renamed, each analysis of that dataset stops, naming the variable, until `jcomplete()` is run again with the current names or the setting is cleared. `jcomplete(on)` and a preview of the set filter stop in the same way, and the status display (`jcomplete()`) names the variable in place of the complete-case count.

Usage

```
jcomplete(
  data,
  ...,
  preview = FALSE,
  console = FALSE,
  non.deletes = FALSE,
  clear.all = FALSE
)
```

Arguments

data	A data frame. If omitted, uses the default set by `juse()`. Instead of variable names, the call may carry one of three special values: the bare word `off` deactivates the setting but remembers the variables, `on` reactivates it, and `NULL` clears it entirely. Each acts on the `juse()` default dataset when `data` is omitted (`jcomplete(off)`, `jcomplete(NULL)`), or on the named dataset when it is given (`jcomplete(d, off)`, `jcomplete(d, NULL)`). With no default set, `jcomplete(NULL)` clears the one dataset that carries a setting, and asks you to name one when several do. A setting is stored under its dataset's name, so the named `off` and `NULL` still work after that dataset has been removed, or its name given to something that is not a data frame; `on` is refused there, since the setting cannot be checked without its dataset. Call with no arguments to check the current status.
...	Unquoted variable names to include in the listwise check.
preview	Logical. If `TRUE`, open a viewer (RStudio data tab) showing the rows the listwise filter will drop, with a leading `Row` column (original data position) and a trailing `DeletionCheck` flag (1 = the row will be dropped). May be used on its own to preview the already-set filter without re-listing the variables (`jcomplete(preview = TRUE)`). Default `FALSE`.
console	Logical or numeric. Print the dropped rows to the console. `TRUE` prints the first 10; a number prints that many. Independent of `preview`: on its own it prints to the console without opening the viewer; combine with `preview = TRUE` to get both. The console listing is always limited to the dropped rows so it cannot flood the console. Default `FALSE`.
non.deletes	Logical. If `TRUE`, the viewer shows every case (with `DeletionCheck` marking which will drop) rather than only the dropped rows. Affects the viewer only; the console listing stays deleted-rows-only. Default `FALSE`.
clear.all	Logical. If `TRUE`, clears the jcomplete setting on every dataset; use on its own, `jcomplete(clear.all = TRUE)`. This is the same grammar as `jsubset()` and the registration functions (`jdummy()`, `jnumeric()`, `jcount()`, `jlikert()`): `NULL` clears one dataset, `clear.all = TRUE` clears them all.

Value

Invisibly returns `NULL`. When a preview is requested, invisibly returns the previewed data frame instead, so it can be captured (e.g. `jcomplete_rows <- jcomplete(preview = TRUE)`).

See Also

jstats for the package overview, workflow conventions, and complete function listing.

Examples

```
juse(community)
jcomplete(Income, Education, Age)
jcomplete(community, Income, Education, Age)         # Explicit dataset
jcomplete(data = community, Income, Education, Age)  # ... given by name
jdesc(Age)                     # Uses only complete cases on those 3 vars
jcomplete(Income, Education, Age, preview = TRUE)  # Set and preview together
```

```
jcomplete(preview = TRUE)      # Preview the already-set filter (viewer)
jcomplete(preview = TRUE, non.deletes = TRUE)  # Viewer shows all cases
jcomplete(console = 10)        # Console only -- first 10 dropped rows
jcomplete(preview = TRUE, console = 25)        # Viewer and console
jcomplete(off)                 # Deactivate
jcomplete(on)                  # Reactivate
jcomplete(community, off)      # Deactivate on a named dataset
jcomplete(community, on)       # ... and reactivate it
jcomplete()                    # Check status
jcomplete(NULL)                # Clear the default dataset's setting
jcomplete(community, NULL)     # Clear a named dataset's setting
jcomplete(clear.all = TRUE)    # Clear every dataset's setting

# juse(NULL) clears the default dataset -- not normally needed. You'd
# clear a default or registration only to undo a mistake, or -- as in
# this example -- to reset state for testing.
juse(NULL)
```

`jconvert` *Convert missing-value declarations between formats*

Description

`jconvert()` provides a single entry point for changing how declared missing values are represented on the columns of a data frame already in memory. Four target formats are supported: SPSS-style (`na_values` on `haven_labelled_spss`), Stata-style (lowercase `tagged_na` on `haven_labelled`), SAS-style (uppercase `tagged_na` on `haven_labelled`), and base R (declarations stripped, declared cells converted to plain `NA`). The haven package and its documentation call these user-defined missing values.

Usage

```
jconvert(
  data,
  ...,
  to = NULL,
  vars = NULL,
  missing.notice = TRUE,
  modify = FALSE
)
```

Arguments

`data`	A data frame, or omitted to use the `juse()` default.
`...`	Optional unquoted variable names, straight after the data frame, as in every data-first function: `jconvert(d, Income, Education, to = "stata")`. When supplied, only the listed variables are scanned. Mutually exclusive with `vars`.

`to` One of `"baseR"`, `"spss"`, `"stata"`, or `"sas"` (any capitalization is accepted), always named: `to = "stata"`. A format name given where a variable goes, as in `jconvert(d, "stata")`, stops and shows the named form. When `to` is left out, `jconvert()` reads the `missing.convention` setting in `joptions`: if the slot is set to `"spss"`, `"stata"`, or `"sas"`, `to` resolves to that value; if the slot is at its `"none"` default, `jconvert()` errors with guidance naming the concrete options. The destructive `"baseR"` target is never auto-resolved – it must always be passed explicitly.

`vars` Alternative scope-by-vector path: a character vector of variable names. Mutually exclusive with `...`. When both `...` and `vars` are empty, `jconvert()` operates on the whole data frame.

`missing.notice` Logical; `TRUE` (default) prints a notification summarizing what was converted – one row per missing value, with its label and what it became – and what was skipped, along with a reminder of how to keep the result. `FALSE` suppresses the message. The notification itself does not consult `joutput()`, because the function reports an action it just performed rather than explaining system behavior; only the keep-the-result reminder follows the tier, and is omitted at `"minimal"`.

`modify` Logical. When `TRUE`, the converted data frame is written back onto the data frame named in the call (or onto the `juse()` default when the data argument is omitted), so no assignment is needed – the recommended workflow, since conversions lost to a forgotten assignment silently change how later analyses treat the affected values. Requires the data frame to be given as a plain name. When `FALSE` (the default), the caller's data frame is untouched; assign the returned data frame to keep the conversions.

Details

The four target formats:

`to = "baseR"` Strip all missing-value declarations and convert declared cells to plain `NA`. For SPSS-style columns (`na_values` / `na_range` on `haven_labelled_spss`), masks declared codes to `NA` and removes the attributes; value labels are preserved so the column can still round-trip through `jsave()` with original labeling. For columns carrying Stata-style missing values (`tagged_na` markers), makes every marker a plain `NA`, labelled or not. A string variable's declared missing values (SPSS declares them directly, as in `MISSING VALUES MARITAL ('UNKNOWN')`) become `NA` the same way. They cannot become Stata-style or SAS-style missing values, which exist only on numeric variables, so `to = "stata"` and `to = "sas"` stop and name such a variable; `jencode` turns it into a numeric one.

`to = "spss"` Convert Stata-style or SAS-style missing values to SPSS-style numeric codes. Each column's distinct letter tags, in letter order, take the `missing.convention.codes` setting in `joptions` (default `-99`, `-98`, `-97`) in order: the lowest letter takes the first code, the next the second, and so on, whatever the letters are – a column carrying `.d`, `.n` and `.r` converts exactly as one carrying `.a`, `.b` and `.c` does. SAS-style (uppercase) tags are case-corrected to Stata-style (lowercase) before the numeric mapping – for round-trip purposes the package treats `.A` and `.a` as the same conceptual marker, and mixed-case columns collapse to a single lowercase marker (SPSS has no parallel uppercase convention). The notification's per-column display shows the original (pre-correction) tag for SAS-corrected columns – e.g. `.A ["Refused"] -> -99` – so the user-visible mapping reflects what was actually in the data on input. A marker

declared only through a value label, with no case carrying it yet, converts like one that cases carry: its label moves to the code and the code is declared missing. A column with more distinct tags than the setting has codes is declared with a missing-value range instead, SPSS's own form for more than three: the codes start at the first `missing.convention.codes` value and run one per tag in the direction the codes run, from the first toward the second – with the defaults, toward zero, so five tags take `-99` through `-95`, declared as the range `-99` to `-95`, each tag's label on its code. The first tags keep their usual codes whenever the setting's codes are consecutive, and the notification shows the range and says why it was used. A setting with a single code (which gives no direction), or one whose range would reach zero, is refused before any data is touched, with the fix. Because the SPSS-to-Stata direction assigns letters by code order, a tag set other than the leading letters comes back as the leading letters (`.d`, `.n`, `.r` return as `.a`, `.b`, `.c`) with the same labels, and codes that do not run from the largest absolute value down – a `missing.convention.codes` setting such as `c(-1, -2, -3)` – come back with their letters in another order; the notification says so whenever a letter would change.

`to = "stata"` Convert SPSS-style numeric codes to Stata-style missing values. Letter tags are assigned by ordering rather than by convention: each column's own declared `na_values` codes are sorted by absolute value descending (ties broken with more-negative-first), then mapped `.a, .b, .c` in that order. Convention codes are NOT consulted for this direction; they only govern the reverse (Stata to SPSS) mapping. Round-trip conversions are not guaranteed to preserve the original numeric codes (e.g. SPSS `c(-1, 9)` -> Stata `.b, .a` -> SPSS `c(-98, -99)` loses the original numbers), but the value labels survive intact and the missingness semantics are preserved. The notification says so whenever the trip back under the `missing.convention.codes` setting would give other codes: "converting Score back to SPSS-style missing values would give its codes -99, -98, not 9, -1." Range-based SPSS missings (`na_range`) are enumerated: Stata-style missing values have no range concept, so the distinct range values present in the column's data, plus any range values carrying a value label, are translated individually. They join the column's discrete `na_values` codes in a single set, sorted and lettered by the same ordering rule. The range rule itself is not preserved – a range value with neither a data occurrence nor a label at conversion time is not translated, so if it first appears in later data it arrives as an ordinary data value. A column whose combined set exceeds 26 values (the `.a`–`.z` alphabet) is refused before any data is touched. A range declaration with no values to translate does not block the conversion: the column still converts, with the empty range declaration dropped and reported. SAS-style (uppercase) tagged columns are case-corrected to Stata-style (lowercase) and counted as converted; columns already fully lowercase are skipped as already in the target form.

`to = "sas"` Identical to `to = "stata"` except that the letters are uppercase (`.A`–`.Z`, SAS's native extended-missing convention): SPSS-style columns are enumerated and mapped to uppercase tags, Stata-style (lowercase) tagged columns are case-corrected to uppercase and counted as converted, and columns already fully uppercase are skipped. Inside R the two tagged forms are the same structure differing only in letter case; note that saving to Stata format (.dta) lowercases uppercase tags again, since the .dta format only supports lowercase letters.

Pre-flight checks for `to = "spss"` include a collision check: if a column's target numeric code (e.g. `-99` for `.a`) is present as genuine data in the column, the call errors before any data is touched; for a column declared with a range, any value inside the range counts. The check covers a marker declared only through a value label as well as one that cases carry. The error message lists every colliding column and the remedy: change the convention codes via `joptions(missing.convention.codes = ...)`. When the same call also meets columns with more tags than codes, it adds the option of

scoping the call with `vars = c(...)` to leave the affected columns out. Atomicity applies to every error mode – the entire `jconvert()` call either succeeds or errors before mutating the data frame.

Pattern A – value labels suggest missingness but no formal declaration. When a numeric column has no formal missing-value declaration but carries value labels matching the package's missing-label wordlist (e.g. `"Refused"`, `"Don't know"`, `"Not applicable"`), `jconvert()` skips the column and surfaces it in the notification with the affected value/label pairs. To formalize these as declared missing values use `jdeclare_missing()`; to leave them as ordinary data, no action is needed. A text variable's labelled values are reported under `to = "spss"` and `to = "baseR"`, and declared with their values in quotes; not under `"stata"` or `"sas"`, whose missing values exist only on numeric variables.

Value

The data frame with the requested conversions applied, returned invisibly. With the default `modify = FALSE`, the caller's data frame is unchanged until the result is assigned back. With `modify = TRUE`, the conversions are also written back onto the caller's data frame, and the returned copy can be ignored.

See Also

`jload` for the load-time strip alternative (`preserve.declarations = FALSE`); `joptions` for setting the default convention and convention codes session-wide.

Examples

```
# community ships with SPSS-style missing values (Income, Education,
# Smoker, Environment1, Environment3), so the conversions run on it
# directly.

# Convert SPSS-style to Stata-style missing values. modify = TRUE
# writes the result back onto df in one step -- the recommended
# workflow.
df <- community
jconvert(df, to = "stata", modify = TRUE)

# Equivalent without modify: assign the returned data frame back
df2 <- jconvert(community, to = "stata")

# Convert to SAS-style missing values (uppercase .A, .B, ...):
df_sas <- jconvert(community, to = "sas")

# Strip the declarations from every applicable variable:
df3 <- jconvert(community, to = "baseR")

# Some variables only: name them after the data frame, then the target
df4 <- jconvert(community, Income, Education, to = "baseR")

# The same, as a character vector:
df5 <- jconvert(community, vars = c("Income", "Education"), to = "baseR")

# Suppress the notification (e.g. inside a script):
```

```
df6 <- jconvert(community, to = "baseR", missing.notice = FALSE)

# The three ways to give the data frame
df7 <- jconvert(community, Income, to = "stata")          # positional
df8 <- jconvert(data = community, Income, to = "stata")   # named
juse(community)                                           # juse() default
df9 <- jconvert(Income, to = "stata")
juse(NULL)

## Not run:
# Convert with target inferred from joptions:
joptions(missing.convention = "spss")
df <- jconvert(df)   # converts any Stata-style columns to SPSS

## End(Not run)
```

---

jcopy *Copy a data frame, carrying its classification registrations*

Description

Copies a data frame to a new name AND clones any classification registrations (jnumeric / jcount / jlikert / jdummy) attached to it, so the copy behaves the same as the original under later analysis calls. A plain assignment (newdata <- mydata) copies the data but not the registrations, because registrations live in a name-keyed session notebook rather than on the data object; jcopy() is the verb that keeps the two together across a rename or copy.

Usage

```
jcopy(data, name, overwrite = FALSE, quiet = FALSE, ...)
```

Arguments

data	The source data frame (unquoted). May be omitted when a juse() default is set, in which case the default frame is the source.
name	The destination name (unquoted) the copy is assigned to. When a single name is given it is read as the destination, not the source.
overwrite	Logical; if FALSE (the default) and the destination name already exists in your environment, an interactive session asks before overwriting. In a script, state it explicitly: `jcopy(mydata, mydata2, overwrite = TRUE)`. Otherwise, if the name already exists in your environment, and the script was run by pasting or with RStudio's Run button, the call stops with an error.
quiet	Logical; if TRUE, suppress the confirmation message.
...	Not used. An input `jcopy()` does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Details

Like jload(), jcopy() cannot see the name on the left of an assignment, so the new name is supplied as an argument. The destination name is unquoted, and a single name is always taken as the destination, with the source coming from the juse() default:

- `jcopy(mydata, newdata)` – copy `mydata` to `newdata`.
- `jcopy(newdata)` – copy the juse() default frame to `newdata`.

Registrations travel only when the source frame carries them; copying an unregistered frame just copies the data. The copy is independent of the original.

Value

Invisibly NULL. Called for its side effect: the copy is assigned into the calling environment under `name`, and its registrations are cloned onto that name.

See Also

jload, jsave, juse

Examples

```
## Not run:
  jdummy(community, Region)        # register a classification on community
  jcopy(community, survey)         # survey carries Region's registration
  jcopy(data = community, survey)  # the data frame given by name

  juse(community)
  jcopy(survey2)                   # copy the default (community) to survey2

## End(Not run)
```

---

jcorr *Bivariate correlation matrix with p values and pairwise N*

Description

Computes pairwise correlations and prints a formatted lower-triangle correlation matrix showing r, p values, and pairwise N for each pair. Supports Pearson (default), Spearman, and Kendall methods. Handles haven-labelled and factor variables with numeric levels. Warns when variables may be categorical rather than continuous.

Usage

```
jcorr(
  data,
  ...,
  method = "pearson",
  subset = NULL,
  variable.id = NULL,
  numeric = NULL,
  categorical = NULL,
  count = NULL,
  value.id = NULL,
  layout = NULL,
  case.processing.detail = NULL,
  digits = NULL,
  print = NULL
)
```

Arguments

data	A data frame.
...	Unquoted variable names within data. A c() of names, bare or quoted, is read as the names.
method	Character. Correlation method: "pearson" (default), "spearman", or "kendall".
subset	An optional unquoted logical expression (e.g. Group == 1) to subset cases for this call only. Applied after jcomplete and jsubset. Does not affect other function calls. Written in R syntax and checked the way jsubset() checks a filter: subset = NOT(Age < 40), for example, is refused with the corrected subset = !(Age < 40) shown (see jsubset for a translation table).
variable.id	Character or NULL. Variable label display mode: one of "both", "names", "labels", "legend", or "legend.bottom". "names" shows variable names only; "both" shows "name: label"; "labels" shows variable labels as the matrix row/column headers (honored even if the matrix grows wide – best for short labels; rerun with a legend mode otherwise); "legend"/"legend.bottom" keep names and print a label legend after the table. NULL (default) defers to joutput()'s variable.id setting. Not a logical.
numeric	Optional. Variables to treat as continuous for this call (the per-call counterpart of jnumeric()), named bare or in quotes, several with c(): numeric = Age. Its only effect in jcorr() is to suppress the structural "seems categorical" caution for those variables; correlations are computed the same way regardless (labelled variables are coerced to numeric either way).
categorical	Not supported by jcorr() yet. Correlation requires numeric variables; supplying categorical raises an error pointing to jcrosstab() for association between categorical variables. (How jcorr() should handle an asserted-categorical variable is a parked design decision.)
count	Optional. Variables to treat as counts for this call (the per-call counterpart of jcount()), named bare or in quotes, several with c(). A count is numeric-like

here, so it behaves like `numeric`: it suppresses the "seems categorical" caution for those variables.

`value.id` Not supported by `jcorr()`. The function does not display value labels, so passing this argument is an error. It exists only to return a clear message rather than report `value.id` as a variable that was not found. Leave at NULL (default).

`layout` Character or NULL. How each correlation cell is laid out when three or more variables are given: `"wide"` (default) puts r and its p-value on one line with N on a second line beneath; `"stacked"` places r, p, and N on three separate lines, giving a narrower table that fits more variables before wrapping. Ignored for a single pair (two variables), which always prints a one-line summary. NULL (default) defers to the `corr.layout` setting in `joptions()` (itself defaulting to "wide").

`case.processing.detail`
Per-call override of the Case Processing Summary detail tier: one of `"none"`, `"totals"`, or `"per_code"`. `NULL` (default) uses the active `joutput()` level default. The Case Processing table itself prints only when a filter or listwise deletion excluded cases; otherwise a one-line N statement takes its place. See `?joutput` (`case.processing`).

`digits` Integer or NULL. Number of decimal places for continuous statistics in the output tables (range 0-7; `digits = 0` prints whole numbers with no trailing decimal point). Does not affect p-values, percentages, or integer quantities (counts, N, degrees of freedom), which keep their own fixed conventions. NULL (default) defers to `joutput()`'s `digits` setting (default 3).

`print` Logical or NULL. `FALSE` prints nothing – no table, no title, no note – and returns the result as every call does; errors and warnings still show. `TRUE` prints even under `joutput("silent")`. `NULL` (the default) prints unless the level is silent. For a script that keeps many results and wants no output: see `joutput`.

Details

A red title identifying the correlation method is printed first, followed by variable labels (if present), then the matrix.

Value

Invisibly returns a list of class `jst_corr` containing: `r` (correlation matrix), `p` (p-value matrix), `n` (pairwise N matrix), `method`, `model_frame` (the analysis data frame used for plotting), `sample_info` (pipeline and missing data counts), `data_name` (the data frame's name, for the lines `jplot()` offers) and `labels`.

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
# With explicit data frame
jcorr(community, Income, Age, WellbeingScore)
```

```
jcorr(community, Income, Age, WellbeingScore, method = "spearman")

# The data frame may also be given by name
jcorr(data = community, Income, Age, WellbeingScore)

# Using juse() default
juse(community)
jcorr(Income, Age, WellbeingScore)
```

`jcount` *Register variables as counts for analysis*

Description

`jcount()` tells jstats to treat one or more variables as count variables (non-negative whole-number tallies). A count is numeric-like – it passes wherever a numeric variable does and shows mean/median in `jscreen` – and additionally carries count semantics: it is the asserted signal behind the count-regression caveat in `jlm` and the routing target for future count-model functions. Unlike the structural guess, jcount accepts counts of any range, including those outside the automatic small-range detection (e.g. a 0-30 victimization count).

Usage

```
jcount(data, ..., remove = FALSE, clear.all = FALSE)
```

Arguments

`data`	A data frame, or omitted to use the `juse` default. `jcount(NULL)` clears the count registrations on the `juse` default frame (or, with no default set, the only frame that carries them; if several do, it asks rather than wiping all). `jcount(data, NULL)` clears that one frame's count registrations. Called with no arguments, `jcount()` lists the session's numeric and count registrations. Registrations are stored under the data frame's name, so it is given as a name (or as a place such as `lst$d`); an expression that builds one is refused, with the two lines that name it first. The `NULL` and `remove = TRUE` forms work on a name whose data frame has since been removed.
`...`	One or more unquoted variable names to register. The registration applies to numeric variables and to haven-labelled variables that hold numbers. Any other type is refused, and none of the call's variables is then registered: convert a factor or a text variable to numbers first with `jencode`.
`remove`	Logical; if `TRUE`, remove the count registration for the named variables instead of adding it.
`clear.all`	Logical; if `TRUE`, clear count registrations on every data frame that carries them.

Details

A variable carries exactly one registered intent at a time, so registering it as a count clears any prior dummy or numeric registration. Registration changes no data and assigns nothing. It is stored for the session, keyed by the data frame's name; save the data frame in R format (.rds) to keep it across sessions.

Value

`invisible(NULL)`. Called for its side effect on the session registration notebook.

See Also

jnumeric, jdummy

Examples

```
df <- data.frame(arrests = c(0, 1, 2, 0, 3, 1, 0, 12),
                 age     = c(21, 34, 45, 29, 51, 38, 26, 60))
jcount(df, arrests)                # treat as a count (here 0-12)
jcount(data = df, arrests)         # the data frame by name
juse(df); jcount(arrests); juse(NULL)  # or the juse() default
jcount(df, arrests, remove = TRUE)
jcount()                           # list all registrations
jcount(df, NULL)                   # clear df's count registrations
jcount(clear.all = TRUE)           # clear every frame's count registrations
```

---

jcrosstab *Cross-tabulation with optional chi-square test of independence*

Description

Produces a cross-tabulation of two categorical variables, showing observed frequencies and row percentages by default. Column percentages, expected frequencies, adjusted standardized residuals, and a chi-square test of independence are available via arguments. Handles haven-labelled, numeric, factor, and character variables. For haven-labelled variables, numeric codes are displayed alongside labels.

Usage

```
jcrosstab(
  formula,
  data,
  chisq = FALSE,
  expected = FALSE,
  row.pct = TRUE,
  col.pct = FALSE,
  residuals = "none",
  subset = NULL,
```

```
  variable.id = NULL,
  value.id = NULL,
  case.processing.detail = NULL,
  digits = NULL,
  categorical = NULL,
  print = NULL,
  ...
)
```

Arguments

formula	A formula of the form `Row ~ Column`, naming plain variables. Transformed terms such as `log(x)` are not supported here – create the variable first (e.g. with `cut()` for binning), then cross-tabulate it.
data	A data frame containing variables referenced in `formula`.
chisq	Logical. If TRUE, prints the chi-square test of independence below the cross-tabulation. For a 2x2 table, two rows are shown – the Pearson chi-square and the Yates continuity-corrected chi-square – matching the rows commercial statistical software reports; the Pearson row is the headline result and is what the returned object carries. Larger tables show the single Pearson result (the correction applies only to 2x2 tables). Under the test, Cramer's V gives its effect size: the square root of the Pearson chi-square over N times one less than the smaller of the table's dimensions, from 0 (no association) to 1. A 2x2 table's line reads "Phi (Cramer's V)", the two being equal there (phi without its sign). When any cell's expected frequency is less than 5, a note under the test gives the number of such cells and the smallest expected frequency. Default is FALSE.
expected	Logical. If TRUE, prints expected frequencies alongside observed, to two decimal places. An expected frequency just under 5 that would round to 5.00 prints as 4.99, so a cell the chi-square note counts as less than 5 never reads as 5. Default is FALSE.
row.pct	Logical. If TRUE (default), shows row percentages.
col.pct	Logical. If TRUE, shows column percentages. Default is FALSE.
residuals	Character. Cell residuals to display: `"none"` (default) or `"adjusted"`. `"adjusted"` adds an `(Adj.Res.)` line to each cell showing the adjusted standardized (Haberman) residual: (observed - expected) divided by its standard error. Under independence these are approximately standard normal, so a value beyond +/-1.96 flags a cell whose count departs from expected at the .05 level. This localizes a significant chi-square to individual cells, and matches the "Adjusted standardized" residual in SPSS CROSSTABS. At `joutput("full")` the residual cells are flagged (`*` past +/-1.96, `**` past the Bonferroni cutoff) and an interpretation note is printed below the table naming both thresholds. Not a logical.
subset	An optional unquoted logical expression (e.g. `Group == 1`) to subset cases for this call only. Applied after jcomplete and jsubset. Does not affect other function calls. Written in R syntax and checked the way `jsubset()` checks a filter: `subset = NOT(Age < 40)`, for example, is refused with the corrected `subset = !(Age < 40)` shown (see `jsubset` for a translation table).

`variable.id` Character or NULL. Variable label display mode: one of `"both"`, `"names"`, `"labels"`, `"legend"`, or `"legend.bottom"`. `"names"` shows variable names only; `"both"` shows `"name: label"`; `"labels"` shows the row/column variable labels (table header and caption; cell value levels follow the value.id mode) – best for short labels; `"legend"`/`"legend.bottom"` keep names and print a label legend after the table. NULL (default) defers to `joutput()`'s `variable.id` setting. Not a logical.

`value.id` Character or NULL. Value-label display mode for both table axes: `"both"` (`"code: label"`), `"values"` (bare code), or `"labels"` (the label, degrading to the bare code where a code has none). `"legend"` and `"legend.bottom"` keep the bare code in the table and print a value-label legend after it (`"legend"` per-table, `"legend.bottom"` consolidated where multiple tables are produced). A no-op for axis variables with no value labels. NULL (default) defers to `joutput()`'s `value.id` setting. Not a logical.

`case.processing.detail` Per-call override of the Case Processing Summary detail tier: one of `"none"`, `"totals"`, or `"per_code"`. `NULL` (default) uses the active `joutput()` level default. The Case Processing table itself prints only when a filter or listwise deletion excluded cases; otherwise a one-line N statement takes its place. See `?joutput` (`case.processing`).

`digits` Integer or NULL. Number of decimal places for continuous statistics in the output tables (range 0-7; `digits = 0` prints whole numbers with no trailing decimal point). Does not affect p-values, percentages, expected frequencies (two places), or integer quantities (counts, N, degrees of freedom), which keep their own fixed conventions. NULL (default) defers to `joutput()`'s `digits` setting (default 3).

`categorical` The variables of the formula to tabulate however many values they hold, named in quotes or without them: `categorical = "Age"`, or `categorical = c(Age, Income)`. A variable with more than 20 values otherwise stops the call, since a variable measured on a scale – an age, a score – makes a table too large to read; the message gives the call with this argument. NULL (default) names none.

`print` Logical or NULL. `FALSE` prints nothing – no table, no title, no note – and returns the result as every call does; errors and warnings still show. `TRUE` prints even under `joutput("silent")`. `NULL` (the default) prints unless the level is silent. For a script that keeps many results and wants no output: see `joutput`.

`...` Not used. An input `jcrosstab()` does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Details

A red "Cross-Tabulation" title is printed first, followed by variable labels (if present), then the table and optional test results.

When the cells carry more than a count – row or column percentages, expected frequencies, residuals – a blank line separates each category's rows from the next, and the Total row from the last category. The columns are set four spaces apart when the table at that spacing fits the message width (`joptions()`'s `message.width`), and two spaces apart when it does not.

Value

Invisibly returns a list of class `jst_crosstab` containing: `observed` (observed frequency table), `expected` (expected frequency table), `adjusted_residuals` (matrix of adjusted standardized residuals), `n` (total N), `model_frame` (the analysis data frame used for plotting), `sample_info` (pipeline and missing data counts), and if `chisq = TRUE`: `chi_square`, `df`, and `p` (the Pearson chi-square), `chi_method` (the test's method string), `cramers_v` (Cramer's V, phi without its sign in a 2x2 table), and for 2x2 tables `chi_square_corrected` and `p_corrected` (the Yates continuity-corrected values).

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
# Cross-tabulation only
jcrosstab(Education ~ Volunteer, data = community)

# The data frame may also be given positionally
jcrosstab(Education ~ Volunteer, community)

# With chi-square test
jcrosstab(Education ~ Volunteer, data = community, chisq = TRUE)

# With expected frequencies and column percentages
jcrosstab(Education ~ Volunteer, data = community,
          expected = TRUE, col.pct = TRUE)

# With adjusted standardized residuals (interpretation note at full output)
jcrosstab(Education ~ Volunteer, data = community, residuals = "adjusted")

# Using juse() default
juse(community)
jcrosstab(Education ~ Volunteer)
jcrosstab(Education ~ Volunteer, chisq = TRUE)
```

---

jdata_dir *Return the configured data folder*

Description

Read-side companion to `joptions(data.dir = ...)`: returns the currently configured data folder as a string, for use in scripts that need the path itself (building a file path, checking existence, cleaning up test files) without reaching into package-internal option names.

Usage

```
jdata_dir(default = ".", ...)
```

Arguments

`default`	Value returned when no data folder is configured. Defaults to `"."` (the working directory).
`...`	Not used. An input `jdata_dir()` does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Details

`joptions()` prints the folder but returns `invisible(NULL)`; `jdata_dir()` returns it as a value. When no folder is configured, the `default` is returned (`"."`, the working directory, by default), so the result drops straight into `file.path`. Pass `default = NULL` to detect the unconfigured state explicitly.

Value

A length-one character string (the configured folder, or `default`); or `default` unchanged when it is `NULL`.

See Also

`joptions` to set the folder; `jload` and `jsave`, which resolve files against it.

Examples

```
## Not run:
joptions(data.dir = "Data")
jdata_dir()                                  # "Data"
f <- file.path(jdata_dir(), "community.rds") # build a path in that folder
if (file.exists(f)) file.remove(f)

jdata_dir(default = NULL)                    # NULL if nothing configured

## End(Not run)
```

---

jdeclare_missing *Declare missing values on one or more variables*

Description

`jdeclare_missing()` declares missing values on one or more variables. Declared missing values are specific data values – typically negative codes such as `-99`, Stata-style tagged markers such as `.a`, or on a text variable words such as `"UNKNOWN"` – that indicate *why* a value is missing (refused, don't know, not applicable, etc.) rather than simply that it is missing. Once declared, those cells are automatically excluded from analyses but remain visible in the data for diagnostic purposes (see `jfreq()`). The haven package and its documentation call these user-defined missing values.

The function operates in declarative mode: what a call mentions, it replaces; what it omits survives. Supplying `codes` replaces the column's discrete-code set; supplying `range` replaces the column's missing-value range; an existing range survives a codes-only call, and existing discrete codes survive a range-only call. A second call therefore replaces, not augments, whichever parts it names – matching SPSS's `MISSING VALUES` and Stata's `mvdecode` semantics, neither of which has an additive form. When prior declarations are dropped, a note lists them so the destructive aspect of the replacement is not silent.

Variables are given either as unquoted names (`jdeclare_missing(df,MathScore, EnglishScore, codes = c(-99))`) or as a character vector via `vars =` (the programmatic form, e.g. `vars = offence_cols`). A multi-variable call applies the same declaration to every named column, all-or-nothing: if any column fails a check, no column is changed. There is deliberately no whole-data-frame default – declaring a code frame-wide would flag it missing even on columns where that value is legitimate data. To declare on every column, pass `vars = names(data)` explicitly.

Usage

```
jdeclare_missing(
  data,
  ...,
  codes = NULL,
  labels = NULL,
  range = NULL,
  vars = NULL,
  convention = NULL,
  missing.notice = TRUE,
  modify = FALSE
)
```

Arguments

`data`	A data frame containing the variable(s).
`...`	The variable(s) to declare missing values on, as unquoted names (e.g. `Income`, or `MathScore, EnglishScore`). Use either `...` or `vars`, not both. All arguments after the variables must be named.
`codes`	The values to declare as missing: numbers, or on a text variable the values themselves in quotes, such as `codes = "UNKNOWN"` (see the Text variables section). Accepts two forms: **Option A (separate codes and labels)** Unnamed numeric vector; labels supplied via the `labels` argument. E.g. `codes = c(-99, -98), labels = "-99=Refused; -98=Don't know"`. **Option C (haven-style named vector)** Named numeric vector; names are the labels. E.g. `codes = c(Refused = -99, `Don't know` = -98)`. On a column that already carries Stata-style or SAS-style missing values, codes may name the markers directly, as quoted strings, e.g. `codes = c(Refused = ".a")` – the string `".a"` names the `.a` marker (or `.A` on a SAS-style column: the letter case you type is accepted either way and canonicalized to the column's convention). A quoted marker is refused on a column with no tagged missing values to label (see the Missing-Values Convention section).

A quoted marker may name one that no case currently carries – useful for FORWARD-DECLARING a label before recoding values into it. The label attaches, nothing in the data changes, and the notification says the marker is not present in the data. Such a label survives a save to Stata format (.dta) and the load back, and `jconvert(to = "spss")` declares its code like any other marker's.

A quoted marker given with no label is a no-op on an already-tagged column: the cells are already missing, so the only act available is naming a marker, and the call reports that it changed nothing. This differs from the numeric form, where a bare code IS the declaration.

`labels` Optional. A quoted string in the form `"value=label; value=label"` pairing labels with codes (Option A only). Must be `NULL` when `codes` is named (Option C). It may also stand alone, with no `codes`: the values named in the string are then the codes declared, so `labels = "-99=Refused; -98=Don't know"` declares and labels both codes in one argument (and `labels = ".a=Refused"` names a marker on a tagged column). A label may be written in quotes, with either mark, right after its equals sign, and a semicolon inside the quotes is then part of the label. When a `range` is in effect (supplied in this call, or already on the column), entries may also name values inside the range: those attach as value labels on the in-range values without becoming discrete declared codes (see the Missing-value ranges section). On a text variable the values are text, as in `labels = "REF=Refused; UNKNOWN=Unknown"`: a value holding an equals sign or a semicolon goes in quotes, and `blank` names the blank cells, as in the maps of `jencode`.

`range` Optional. A length-2 numeric vector declaring a missing-value RANGE (band), e.g. `range = c(-99, -51)` – every value from the first bound through the second is treated as missing. The SPSS parallel is `MISSING VALUES X (-99 THRU -51)`. Bounds may be non-integer, and one bound may be infinite (`c(-Inf, -51)` is SPSS's `LO THRU -51`). A missing-value range can exist only under SPSS convention: combining `range` with the Stata or SAS convention – as a per-call argument or as the `joptions()` setting – or with quoted markers is refused, and the refusal teaches the two-step route (declare the range under SPSS convention, then `jconvert()` the column) when the range covers few enough values to convert. When no convention is selected anywhere, a range declaration stops and asks for `convention = "spss"` on the call. SPSS allows at most ONE discrete code alongside a range, and the check applies to the column's composed result (what this call supplies plus what already survives on the column), so a declaration that a `.sav` file could not hold is refused here rather than at save time. A text variable takes no range.

`vars` Optional. A character vector of variable names, the programmatic alternative to unquoted names in `...` (e.g. `vars = c("Age", "Income")`, or `vars = offence_cols` where `offence_cols` holds the names). Use either `vars` or `...`, not both.

`convention` Optional. One of `"spss"`, `"stata"`, or `"sas"` (any capitalization is accepted); sets the convention for this call on columns that carry no missing-value declaration yet. A column that already carries the other convention's missing values is not overridden: the call stops and offers the two remedies (drop the argument to follow the column's form, or `jconvert()` the column first). When `NULL` (the default), the convention is resolved from the column's existing missing-value declaration (if any), then from the `missing.convention` setting in `joptions`; when

neither supplies one, the call stops with a guided error asking you to choose – the package never infers a convention for a fresh declaration. A `range` requires SPSS convention (see `range`). The SAS convention behaves as the Stata convention with uppercase markers: markers are stored and labeled as `.A-.Z`. A quoted marker's letter case is accepted either way under both tagged conventions; the case written to the column follows the resolved convention. A text variable's missing values are SPSS-style whatever the setting, so `"stata"` and `"sas"` are refused for one.

`missing.notice` Logical. When `TRUE` (the default), the function prints a notification summarizing what was declared, plus a reminder of how to keep the result. Under SPSS convention the notification lists the variable's resulting declaration, not only what the call named: a range or code the variable keeps from an earlier call is marked "already declared", a code with no value label is marked "no label", and a code or range that no case holds is marked "not present in the data" (the mark a mistyped code gets, such as -77 for -99). In a call on several variables that last mark is given only when none of the listed variables holds the value. Set `FALSE` to suppress.

`modify` Logical. When `TRUE`, the declaration is written back onto the data frame named in the call (or onto the `juse()` default when the data argument is omitted), so no assignment is needed – the recommended workflow, since a declaration lost to a forgotten assignment silently poisons later statistics. Requires the data frame to be given as a plain name. When `FALSE` (the default), the caller's data frame is untouched; assign the returned data frame to keep the declaration.

Value

The data frame, with the specified variable(s) updated to carry the declared missing values, returned invisibly. With the default `modify = FALSE`, the caller's data frame is unchanged until the result is assigned back. With `modify = TRUE`, the change is also written back onto the caller's data frame, and the returned copy can be ignored.

Missing-Values Convention

Under SPSS convention, codes are declared as numeric values via the column's `na_values` attribute (haven's representation of SPSS-style missing values). The data cells themselves are unchanged; only the metadata that flags certain values as missing is added.

Under Stata or SAS convention with tagged missing-value input (quoted markers such as `".a"`), the function attaches value labels to the column's existing tagged missing-value markers. This requires the column to already carry tagged missing values – either tagged cells, or markers previously declared through value labels (a marker may be labeled before any cases carry it, so a declaration made early in data collection is complete for later data). The letter case you type is accepted either way and canonicalized to the resolved convention: lowercase markers under Stata convention, uppercase under SAS. Quoted markers against a column with no tagged missing values are refused identically under every convention source: on a plain column the error points at `jrecode()` (which creates tagged cells from numeric codes); on a column carrying SPSS-style declarations it points at `jconvert()`.

Note that on a plain numeric column with plain numeric codes and no `convention` argument, the resolved convention decides the outcome: under SPSS convention the numbers stay in the cells and

are flagged missing; under Stata or SAS convention the matching cells are converted to markers and the numbers leave the data. This is the one place `joptions(missing.convention = ...)` changes what happens to data (see the examples).

Under Stata or SAS convention with numeric input, the function converts matching cells to tagged missing-value markers (SAS convention writes the same letters uppercase). The mapping is ordering-based: codes sorted by absolute value descending, more-negative-first as tie-breaker, then assigned `.a`, `.b`, `.c`, `.d` in that order (`.A`, `.B`, ... under SAS convention). The assignment proceeds independently of the `missing.convention.codes` setting in `joptions` (which only governs the reverse Stata-to-SPSS direction).

The letters are taken from the start of the alphabet whatever the variable already holds, so a code is refused when its letter is one the variable already carries, in its cells or as a labeled value: converting `-99` to `.a` on a variable whose `.a` cells mean "Skipped" would put two kinds of missing data on one marker. The message gives two `jrecode()` calls that run: one recodes the codes to markers the variable does not use, which keeps the kinds distinct; the other recodes them to the markers they would have taken, for kinds that mean the same thing.

Missing-value ranges

A range declares a whole band of values missing at once – the form commercial statistical software uses when a study's missing-value codes share a band (e.g. every code from -99 through -51). SPSS accepts at most three discrete missing values, OR a range, OR a range plus one discrete value; `jdeclare_missing()` enforces the same rule on the composed result of each call, so a declaration is refused at the moment it becomes illegal rather than when `jsave()` later refuses the file. A range-only call is complete in itself (`jdeclare_missing(df, X, range = c(-99, -51))`).

Values inside the band may carry value labels: with a range in effect, `labels` entries that match no discrete code but fall inside the band attach as ordinary value labels on those values. They do not become discrete declared codes – the band already covers them – but analysis output that breaks out in-range values can then show their meanings. Because a range replaces the column's existing range and existing discrete codes survive a range-only call, a column already carrying two or more discrete codes cannot take a range in the same declaration; the call is refused with the surviving codes named. When the new range no longer covers all of the old one, a note names the range as it was and counts the cases that are no longer missing.

Text variables

On a text variable the codes are the values themselves, in quotes: `jdeclare_missing(df, Marital, codes = "UNKNOWN")`, or with labels, `codes = c(Refused = "REF")` or `labels = "REF=Refused"`. They are declared SPSS-style – the form SPSS's `MISSING VALUES` gives a string variable, and the only form a text value has – under every `missing.convention` setting and with none selected, so no convention needs choosing first. Under a Stata or SAS setting a note says so; `convention = "stata"` or `"sas"` and a `range` are refused for text. At most three values may be declared, the SPSS limit on a string variable.

A value matches the cells that hold it exactly as stored: letter case and outer spaces count, as in SPSS. After declaring, a note names any values that differ from a declared one only in case or outer spaces (`"Unknown"` beside `"UNKNOWN"`), with their counts, and gives the call that declares them too – naming the values already declared, since a call replaces a variable's declaration – or, past three values, points to `jencode`. `codes = ""` declares every blank cell missing – empty, or holding only

spaces or tabs – as SPSS declares a blank string missing; otherwise jstats counts blank cells as valid values and never assumes they are missing.

Numbers given for a text variable are refused with the call to run, the numbers in quotes. A call naming text and numeric variables together declares numbers on the numeric ones or words on the text ones, and refuses the others with the call to run. To keep the values in an SPSS file, each must be 8 characters or fewer: `jsave()` refuses a longer one, which the file would cut short.

Mixed conventions and file export

A single data frame may carry columns with SPSS-style and columns with Stata-style missing values. In-memory analysis and display tolerate the mix without issue (each column renders in its native form). The constraint shows up at file-export time: `.sav` cannot represent Stata-style missing values; `.dta` cannot represent SPSS-style `na_values` declarations; `.xpt` can represent neither form. `jsave()` pre-flights the DF against the destination format and errors with a pointer to `jconvert()` when the mix is incompatible. The post-declaration mismatch notice emitted at the bottom of this function's output exists to alert you early if a declaration ends up out of step with the rest of its DF.

See Also

jrecode, jconvert, joptions, jstats

Examples

```
# A fresh declaration needs a chosen missing-value convention; the
# package never infers one. Choose SPSS convention for these examples:
joptions(missing.convention = "spss")

# clinic$MoodRating arrives "dirty": -99/-98 sit in the data as
# ordinary numbers (the state after a CSV or Excel import), so summary
# statistics are poisoned until the codes are declared missing.
df <- clinic
jdesc(df, MoodRating)        # mean dragged far down by -99/-98

# SPSS form: declare -99 and -98 as missing, with labels. modify = TRUE
# writes the declaration back onto df in one step -- the recommended
# workflow.
jdeclare_missing(df, MoodRating,
             codes  = c(-99, -98),
             labels = "-99=Refused; -98=Don't know",
             modify = TRUE)
jdesc(df, MoodRating)        # codes now excluded as missing

# Equivalent without modify: assign the returned data frame back
df2 <- clinic
df2 <- jdeclare_missing(df2, MoodRating,
                    codes  = c(-99, -98),
                    labels = "-99=Refused; -98=Don't know")

# Equivalent using named codes (one step instead of codes + labels)
df3 <- jdeclare_missing(clinic, MoodRating,
                    codes = c("Refused" = -99, "Don't know" = -98))
```

```
# The data frame may also be given by name, or set once with juse()
df3 <- jdeclare_missing(data = clinic, MoodRating, codes = c(-99, -98))
juse(clinic)
df3 <- jdeclare_missing(MoodRating, codes = c(-99, -98))
juse(NULL)

# A missing-value RANGE: every value from -99 through -51 is missing.
# SPSS parallel: MISSING VALUES MoodRating (-99 THRU -51).
df4 <- jdeclare_missing(clinic, MoodRating, range = c(-99, -51))

# Range plus labeled values inside the band, on several variables at
# once. The same declaration lands on every named column,
# all-or-nothing.
## Not run:
jdeclare_missing(mydata, vars = c("Theft", "Assault", "Burglary"),
             range  = c(-99, -51),
             labels = "-99=Refused; -61=Not applicable",
             modify = TRUE)

## End(Not run)

# Stata-style: label Stata-style missing-value cells. The jrecode() call
# turns the literal codes into tagged cells; jdeclare_missing() labels them
# by naming the markers as quoted strings.
df5 <- clinic
df5$Mood2 <- jrecode(df5, MoodRating,
                     map = "-99=.a; -98=.b; else=copy",
                     convention = "stata")
jdeclare_missing(df5, Mood2,
             codes = c("Refused" = ".a", "Don't know" = ".b"),
             modify = TRUE)

# A text variable: the values themselves, in quotes, declared SPSS-style
# under every setting. codes = "" declares the blank cells.
df8 <- data.frame(Marital = c("Married", "UNKNOWN", "Single", "",
                              "UNKNOWN", "Married"))
df8 <- jdeclare_missing(df8, Marital, codes = c(Unknown = "UNKNOWN"))
df8 <- jdeclare_missing(df8, Marital, codes = c("UNKNOWN", ""))
jfreq(df8, Marital)

## Not run:
# The same neutral call -- plain numeric codes, no convention argument,
# plain column -- forks on joptions(missing.convention = ...):
joptions(missing.convention = "spss")
df6 <- jdeclare_missing(clinic, MoodRating, codes = c(-99))
# -99 stays in the cells, flagged as missing (SPSS-style declaration)

joptions(missing.convention = "stata")
df7 <- jdeclare_missing(clinic, MoodRating, codes = c(-99))
# -99 cells become the .a marker; the number -99 leaves the data

## End(Not run)
```

jdesc *Descriptive statistics for one or more variables*

Description

Computes basic descriptive statistics (N, non-missing, min, max, mean, SD) for one or more variables in a data frame. Prints a formatted table and invisibly returns the underlying results as a data frame.

Usage

```
jdesc(
  data,
  ...,
  by = NULL,
  subset = NULL,
  variable.id = NULL,
  numeric = NULL,
  categorical = NULL,
  count = NULL,
  value.id = NULL,
  case.processing.detail = NULL,
  digits = NULL,
  print = NULL
)
```

Arguments

`data`	A data frame, or a single variable: a column of a data frame or a numeric vector.
`...`	Unquoted variable names within `data`. None when `data` is a single variable. A `c()` of names, bare or quoted, is read as the names.
`by`	An optional unquoted grouping variable name. When provided, descriptives are computed separately for each group, with a separate titled table per dependent variable. Each table's `Total` is the group's size and `Non_missing` the cases in it with a value on that variable, the same two columns as the ungrouped table. A case with a missing value on the grouping variable belongs to no group and so to no table; the Case Processing Summary reports those cases on a `by =` row (shown only when there are any), and its N counts the cases that have a group. The grouping variable cannot also be one of the variables described.
`subset`	An optional unquoted logical expression (e.g. `Group == 1`) to subset cases for this call only. Applied after jcomplete and jsubset. Does not affect other function calls. Written in R syntax and checked the way `jsubset()` checks a filter: `subset = NOT(Age < 40)`, for example, is refused with the corrected `subset = !(Age < 40)` shown (see `jsubset` for a translation table).

variable.id — Character or NULL. Variable label display mode: one of "both", "names", "labels", "legend", or "legend.bottom". "names" shows variable names only; "both" shows "name: label"; "labels" shows each variable's label in place of its name (in the descriptives table; for grouped output, as the per-variable caption and the grouping-variable column header) – best for short labels; "legend" and "legend.bottom" keep names and print a label legend after the table. NULL (default) defers to joutput()'s variable.id setting. Not a logical.

numeric — Optional. Variables to treat as continuous for this call (the per-call counterpart of jnumeric()), named bare or in quotes, several with c(): numeric = Age. Its only effect in jdesc() is to suppress the structural "seems categorical" descriptive caution for those variables; the descriptives themselves are computed the same way regardless.

categorical — Not supported by jdesc() yet. jdesc() always computes numeric descriptives; supplying categorical raises an error pointing to jfreq() for a categorical summary. (How jdesc() should handle an asserted-categorical variable is a parked design decision.)

count — Optional. Variables to treat as counts for this call (the per-call counterpart of jcount()), named bare or in quotes, several with c(). A count is numeric-like here, so it behaves like numeric: it suppresses the "seems categorical" caution for those variables.

value.id — Character or NULL. Value-label display mode for the grouped descriptive headers (the by-group rows): "both" ("code: label"), "values" (bare code), or "labels" (the label, degrading to the bare code where a code has none). "legend" and "legend.bottom" keep the bare code in the table and print a value-label legend after it ("legend" per-table, "legend.bottom" consolidated where multiple tables are produced). A no-op for grouping variables with no value labels, and for ungrouped calls. NULL (default) defers to joutput()'s value.id setting. Not a logical.

case.processing.detail — Per-call override of the Case Processing Summary detail tier: one of "none", "totals", or "per_code". NULL (default) uses the active joutput() level default. The Case Processing table itself prints only when a filter or listwise deletion excluded cases; otherwise a one-line N statement takes its place. See ?joutput (case.processing).

digits — Integer or NULL. Number of decimal places for continuous statistics in the output tables (range 0-7; digits = 0 prints whole numbers with no trailing decimal point). Does not affect p-values, percentages, or integer quantities (counts, N, degrees of freedom), which keep their own fixed conventions. The Min and Max columns are values of the variables rather than statistics: each variable shows as many decimal places as its own data carry, up to this many, so a variable measured in whole numbers prints 0 and 75 beside another's 4.8 and 9.7, and the two columns are aligned on the decimal point. NULL (default) defers to joutput()'s digits setting (default 3).

print — Logical or NULL. FALSE prints nothing – no table, no title, no note – and returns the result as every call does; errors and warnings still show. TRUE prints even

under `joutput("silent")`. `NULL` (the default) prints unless the level is silent. For a script that keeps many results and wants no output: see `joutput`.

Details

Output is structured consistently with `jfreq()`: a single red "Descriptive Statistics" title is printed first, followed by the default-data note (if a juse() default was used), any pipeline messages, and then either the Case Processing table (when a filter is active) or a one-line statement of the N – for several variables, the size of the variable pool and how many cases are complete on all of them. One shared table follows, a row per variable. Variable labels are not printed in the table; `variable.id` puts them in the row names or in a legend under the table.

Summarizes numeric, haven-labelled, logical, numeric-coded factor, and numeric-looking character variables. Variables that cannot be summarized — text factors, text character variables, and date/time variables — are skipped with a warning directing the user to `jfreq()` (date/time variables are not supported here). When every requested variable is unsummarizable, jdesc() stops with an error. Also accepts a simple numeric vector. Supports grouped descriptives via the by parameter.

Given a single variable instead of a data frame, as in `jdesc(community$Age)`, jdesc() describes that variable. A column of a data frame is described in its frame, exactly as `jdesc(community, Age)` describes it: the frame's `jsubset()` and `jcomplete()` settings and its registrations apply. Any other vector is described on its own. A second variable, by, and a `subset` condition naming another variable need the data frame, and each stops with that form of the call.

With a `juse()` default set, a name the default data frame has is that frame's variable, as in every jstats function, even when a separate object of the same name exists in the workspace: `jdesc(Age)` describes the default frame's Age, and a line under "Using default data frame" says so. A separate object is read only when the default frame has no variable of that name, or when no default is set.

With by, the case-processing summary counts the cases that have a group. Where it is a single line rather than a table (at the minimal output level, for example), that line counts "Grouped Cases" once the grouping variable has excluded any.

A haven-labelled variable is summarized by its codes, like any numeric variable. One whose codes look like categories gets a warning that the statistics may not be meaningful, with `jfreq()` as the alternative.

Value

Invisibly returns a list of class `jst_desc` containing: `descriptives` (data frame of statistics, or NULL for grouped output), and `sample_info` (pipeline and missing data counts). Also prints a formatted table to the console.

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
# With explicit data frame
jdesc(community, Age)
jdesc(community, Income, Age, WellbeingScore)
jdesc(community, WellbeingScore, by = Volunteer)
```

```
# The data frame may also be given by name
jdesc(data = community, Age)

# Using juse() default
juse(community)
jdesc(Age)
jdesc(Income, Age, WellbeingScore)
jdesc(WellbeingScore, by = Volunteer)

# With a vector directly
jdesc(community$Age)
```

---

jdummy *Register categorical variables for dummy coding in regression*

Description

jdummy() registers a categorical variable so that jlm() automatically expands it into dummy (indicator) variables when it appears in a regression formula. The original data frame is never modified. Several variables can be registered in one call; the ref setting then applies to each of them.

Registrations are stored per dataset, so switching juse() between datasets preserves each dataset's registrations independently.

Usage

```
jdummy(
  data,
  ...,
  ref = "auto",
  show = FALSE,
  remove = FALSE,
  clear.all = FALSE,
  max.categories = 20L
)
```

Arguments

data	A data frame, or omit to use the juse() default. jdummy(NULL) clears the dummy registrations on the juse() default data frame (or, with no default set, the only frame that carries them; if several do, it asks rather than wiping all). Registrations are stored under the data frame's name, so it is given as a name (or as a place such as lst$d); an expression that builds one, such as a subset, is refused, with the two lines that name it first. For the same reason jdummy(data, NULL) and remove = TRUE work on a name whose data frame has since been removed.

...	One or more unquoted variable names to register. The variable may be haven-labelled, numeric, logical, character, or a factor: for a factor the levels are the categories, in level order, and a level with no cases is left out; for a text variable the distinct values are the categories, in alphabetical order, and cells with no text (empty, or holding only spaces or tabs) are one more category, <blank>, placed last. Omit (along with data) to display all current registrations. A lone NULL in the variable slot – jdummy(data, NULL) – clears that frame's dummy registrations.
ref	The reference category (excluded from the regression model). Can be a numeric code, a quoted label name, first, last, or auto (the default). Under auto, a variable with three or more categories takes its first (lowest) category as the reference. A two-category variable is parameterized so the category representing presence is the one modeled: if the value labels are a recognized yes/no pair (yes/no, y/n, true/false, t/f, present/absent, success/failure; capitalization does not matter), the affirmative category is modeled and the negative category is the reference – the same rule jlogistic() uses to encode its outcome. Otherwise, if one of the two codes is 1, that category is modeled (so a 1 = Yes / 2 = No variable models Yes, and a 0/1 variable models 1). A text variable holding one word or nothing ("Y" or blank) models the word, with <blank> as the reference. Applied to every variable named in the call; to use different reference categories, register the variables in separate calls. One value only. Every variable is checked before any is registered, so a call that stops – a reference one of the variables does not have, for example – registers none of them. When the reference was chosen by auto, the printed "Reference category:" line ends with (default; change with ref =); a reference you name carries no such tag.
show	Logical. If TRUE, prints the dummy coding scheme table showing the pattern of 0s and 1s. Default is FALSE.
remove	Logical. If TRUE, removes the registration for the specified variable(s). Default is FALSE.
clear.all	Logical. If TRUE, clears dummy registrations on every data frame that carries them. Default is FALSE.
max.categories	Integer. Maximum number of categories a variable may have to be dummy-coded; a variable with more raises an error. Raise it to dummy-code a higher-cardinality variable. Default 20L.

Value

Invisibly returns NULL. Called for its side effect.

See Also

jstats for the package overview, workflow conventions, and complete function listing.

Examples

```
juse(community)
jdummy(Region)                      # Register, first category as reference
jdummy(community, Region)           # The same, naming the data frame
```

```
jdummy(data = community, Region)     # ... or naming it as data =
jdummy(Region, Education)            # Register several at once
jdummy(Region, ref = "last")         # Last category as reference
jdummy(Region, ref = 4)              # Reference by numeric code
jdummy(Region, ref = "East")         # Reference by value label
jdummy(Region, show = TRUE)          # Show coding scheme
jdummy(Region, show = "all")         # Full scheme (for many categories)
jdummy()                             # Show all registrations
jdummy(Region, remove = TRUE)        # Remove one registration
jdummy(community, NULL)              # Clear community's dummy registrations
jdummy(NULL)                         # Clear the default frame's (or ask)
jdummy(clear.all = TRUE)             # Clear every frame's dummy registrations

# juse(NULL) clears the default dataset -- not normally needed. You'd
# clear a default or registration only to undo a mistake, or -- as in
# this example -- to reset state for testing.
juse(NULL)
```

---

jencode *Encode a text or factor variable as labelled numbers*

Description

Converts a character (text) or factor variable into numeric codes, attaching the original words as value labels so every later table still shows the words. With no map, codes are assigned alphabetically (a factor's levels keep their own order) and the assignment is printed; with a map, you choose the numbers. Numbers stored as text ("34") always convert to their own value, never to a rank.

Usage

```
jencode(data, var, map = NULL, labels = NULL, convention = NULL, ...)
```

Arguments

data	A data frame containing the variable to encode. Can be omitted if a default data frame has been set with juse().
var	The text or factor variable to encode (unquoted name; a quoted name is accepted). Character variables and factors are accepted; numeric, logical, and date/time variables are refused with a message naming the right tool (for numeric variables, that is jrecode()).
map	Optional. A single string of semicolon-separated rules, each word=number: for example "Bail=1; Parole=2; Remand=3". Matching against the data is case-sensitive, after outer spaces are trimmed on both sides (a note reports any cells that matched only after trimming). Words containing a semicolon must be double-quoted ('"Not applicable; other"=3'); apostrophes need no quoting: map = "Yes=1; Don't know=8" works as typed.

Special left-hand sides: `blank=<number>` gives empty cells their own code. With no map, blanks are left missing and a note shows the blank rule to use; with a map, blanks count as unmapped, so a map that neither names them nor carries an `else` rule stops with the same incomplete-map error. Special targets: `else=NA` converts every unmapped word to system missing; `else=.a` (through `.z`) converts them to a tagged missing value under the Stata or SAS convention (see `convention`). `else=copy` is refused: words cannot be kept in a numeric column. A word may also be sent to the word `missing` – your working convention's own missing form, declared automatically under the SPSS convention (see `jrecode()`'s `map` for the full rule); `blank=missing` composes the two keywords. Under the SPSS convention the code the keyword uses must be free: if the map also assigns it as an ordinary value, or a number stored as text is kept at that same value, the call stops rather than sweep those cases into missingness. Under the Stata or SAS convention the keyword always uses the first marker (`.a`, or `.A`), so a map that also assigns that marker itself stops rather than merge two missing categories onto one marker.

By default an incomplete map is an error that names the unmatched words (nothing is dropped silently); add an `else` rule to sweep the remainder deliberately, and the note then names the words it swept.

`labels` Optional. A single string of semicolon-separated `number=Label` pairs overriding the automatic labels: for example `"1=Low; 2=Medium; 3=High"`. If omitted, each word becomes the label of its own code. When several words collapse to one code, no label can be chosen automatically and a note directs you here or to `jrelabel()`. A label may be written in quotes, with either mark, right after its equals sign, and a semicolon inside the quotes is then part of the label: `"1=Yes; 9=\"Refused; not asked\""`. An unquoted label reads as it is written.

`convention` Optional. One of `"spss"`, `"stata"`, `"sas"`, or `NULL` (default); any capitalization is accepted. Controls whether missing-value markers (`.a` through `.z` or `.A` through `.Z`) are accepted as map targets. Marker letters are matched case-insensitively; the stored markers take the convention's letter case (lowercase Stata-style under `"stata"`, uppercase SAS-style under `"sas"`). Also decides what the `missing` keyword produces. Inert when neither a marker nor `missing` appears in the map.

`...` Not used. An input `jencode()` does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

When `NULL`, the convention is resolved from the `missing.convention` setting in `joptions`; if that is also unset, the call stops with a guided error asking you to choose – the package never infers a convention. Most users set the convention once at the top of a session via `joptions()` (or in their `.Rprofile`) rather than supplying this argument on every call. See `?joptions` for details.

Details

The SPSS parallel. `AUTORECODE` is the SPSS command for this job. Given a text variable it assigns consecutive numbers to the distinct values in alphabetical order, stores the original text as value labels, and writes the result to a new variable. Before:

```
Status:  "Parole"  "Bail"  "Remand"  "Bail"
```

After `AUTORECODE VARIABLES=Status /INTO StatusR.`:

```
StatusR: 2  1  3  1
with value labels  1 "Bail"  2 "Parole"  3 "Remand"
```

`jencode()`'s automatic mode does exactly this, and additionally prints the assignment listing plus a ready-made `map` call so an ordered set (Low / Medium / High) can be renumbered deliberately rather than accepted alphabetically.

A factor is text with a declared order. R's `factor()` stores categories as level text in a set order, so `jencode()` accepts a factor as it accepts a text variable, with two differences that honor the declaration: the codes follow the level order rather than the alphabet (the same thing for a factor built without `levels =`, whose levels are alphabetical), and a level with no cases keeps its code and label, tagged `(no cases)` in the listing. An ordered factor (`ordered = TRUE`) has already declared its order, so the rerun-with-a-map suggestion is not printed for it. Levels that are all numbers convert by face value, like numbers stored as text. With a `map`, the rules name the level text exactly as they name words. This is also the route from a factor to `jrecode()` and `jrelabel()`, which work on numeric variables only, and to `jdeclare_missing()`, which takes numeric and text variables but not a factor.

Numbers stored as text convert by face value. Here the two part company. `AUTORECODE` treats "34" as just another string and renumbers it to its alphabetical rank; `jencode()` always converts a number stored as text to its own value ("34" becomes 34, never 2), because a column of ages that arrives as text should leave as ages. When every cell is numeric text, no value labels are attached and a note says so.

Repair mode. A map containing only an `else` rule – `map = "else=NA"` – is the one-line repair for a poisoned column (numbers plus stray word codes plus blanks, the shape a spreadsheet import often produces): the numbers are kept at face value and every word and blank is swept to missing. When a kept value looks like a coded missing value – by magnitude, or because a swept word such as "Refused" is evidence the column carried missing-value codes – a note suggests declaring it with `jdeclare_missing()` rather than losing it.

Blanks are counted separately. An empty cell ("") is neither a word nor an NA. In automatic mode blanks are left missing, with a note showing the `blank=` rule; with a map they must be named or swept by an `else` rule, and mapping `blank=0` (or any code) gives them their own category, which matters in field data where a blank often means "No". Every map a note offers after an automatic call names the blank cells (and any declared missing strings, below), so each offered call runs as printed.

A map word the data do not hold. A map may name a word no case holds, to label a category ahead of time; a note says so at the full output level. When an `else` rule sent a word of the data to missing in the same call, the note is shown at every level, under the note naming the swept word: together the two usually mean a mistyped map word (`"Parol=2; else=NA"` sends every "Parole" to missing).

Declared missing values on a text variable. A file from SPSS can declare some of a string variable's values missing (`MISSING VALUES MARITAL ('UNKNOWN')`). In automatic mode those cells are left missing rather than numbered as categories, and a note gives the map that keeps them declared (`UNKNOWN=missing`). The word `missing` produces one missing value, so two declared strings sent

to it share it; to keep them apart, give each a number of its own in the map and declare those numbers with jdeclare_missing. With a map they are words like any other: name them, or the call stops and lists them.

The variable label from the original variable is carried across automatically with "(encoded)" appended; if there is none, the variable name is used instead.

Value

A haven_labelled numeric vector with the encoded values and (unless overridden) the original words as value labels. The function returns the values rather than changing data: assign the result to a column to keep it. Two assignment forms do that job:

- Preserving (recommended): MyData$StatusR <- jencode(MyData, Status) adds a new column and keeps the original text, so the two can be compared with jfreq() before the text is retired.
- Overwrite: MyData$Status <- jencode(MyData, Status) replaces the text in place. Nothing is wrong with this once the encoding is verified, but the original words in the data are gone.

SPSS users will recognize the pair: RECODE ... INTO new. is the preserving form and bare RECODE the overwrite, and SPSS's own text-to-numbers command (AUTORECODE, below) offers only the preserving form.

See Also

jrecode for changing numeric values, jrelabel for value labels, jdeclare_missing for declaring missing-value codes, jfreq for checking an encoding landed correctly.

Examples

```
MyData <- data.frame(
  Status = c("Parole", "Bail", "Remand", "Bail"),
  Answer = c("Yes", "No", "Don't know", ""),
  AgeTxt = c("34", "41", "-99", "Refused"),
  stringsAsFactors = FALSE
)

# Automatic mode: alphabetical, listing printed, words become labels
MyData$StatusR <- jencode(MyData, Status)

# The data frame may also be given by name, or set once with juse()
MyData$StatusR <- jencode(data = MyData, Status)
juse(MyData)
MyData$StatusR <- jencode(Status)
juse(NULL)

# Map mode: choose the numbers; apostrophes need no quoting
MyData$AnswerR <- jencode(MyData, Answer,
                          map = "Yes=1; No=0; Don't know=8; blank=9")

# Repair mode: keep the numbers at face value, sweep words and blanks
```

```
MyData$Age <- jencode(MyData, AgeTxt, map = "else=NA")

# A factor: level order kept; an empty level keeps its code and label
MyData$Sev <- factor(c("Low", "High", "Low", "High"),
                     levels = c("Low", "Medium", "High"))
MyData$SevR <- jencode(MyData, Sev)
```

`jfreq` *SPSS-like frequency tables for categorical variables*

Description

Prints an SPSS-style frequency table (Freq, Total %, Valid %, Cum. %) for each variable supplied. Designed for use with unquoted variable names, and also accepts a plain vector.

Usage

```
jfreq(
  data,
  ...,
  subset = NULL,
  variable.id = NULL,
  value.id = NULL,
  missing.detail = NULL,
  case.processing.detail = NULL,
  print = NULL
)
```

Arguments

`data`	A data frame, or a single variable: a column of a data frame or a vector.
`...`	Unquoted variable name(s) within `data`. None when `data` is a single variable. A `c()` of names, bare or quoted, is read as the names.
`subset`	An optional unquoted logical expression (e.g. `Group == 1`) to subset cases for this call only. Applied after jcomplete and jsubset. Does not affect other function calls. Written in R syntax and checked the way `jsubset()` checks a filter: `subset = NOT(Age < 40)`, for example, is refused with the corrected `subset = !(Age < 40)` shown (see `jsubset` for a translation table).
`variable.id`	Character or NULL. Variable label display mode: one of `"both"`, `"names"`, `"labels"`, `"legend"`, or `"legend.bottom"`. `"names"` shows variable names only; `"both"` shows `"name: label"`; `"labels"` uses each variable's label as its table caption (best for short labels); `"legend"` prints a label legend under each variable's own table; `"legend.bottom"` prints one consolidated legend after all tables. NULL (default) defers to `joutput()`'s `variable.id` setting. Not a logical. (Replaces the former inline Type/label block.)

`value.id` Character or NULL. Value-label display mode for the frequency-table valid rows: `"both"` (`"code: label"`), `"values"` (bare code), or `"labels"` (the label, degrading to the bare code where a code has none). `"legend"` and `"legend.bottom"` keep the bare code in the table and print a value-label legend after it (`"legend"` per-table, `"legend.bottom"` consolidated where multiple tables are produced). A no-op for variables with no value labels. NULL (default) defers to `joutput()`'s `value.id` setting. Not a logical.

`missing.detail` Character or NULL. How much of a declared missing-value RANGE the Missing block spells out: one of `"totals"` (collapse the whole band into a single row), `"per_code"` (one row per observed in-band value, at most 10, with the remainder gathered into one line at the foot of the block), or `"all"` (every observed in-band value, no cap). NULL (default) defers to the `missing.detail` setting in `joptions`, whose own default is `"per_code"`. Declared discrete codes always print in full at every setting; the cap applies only to values reached by a range. Has no effect on a variable with no declared range.

`case.processing.detail` Accepted for API symmetry. jfreq's Case Processing Summary is top-table only (no missing-data breakdown), so this argument has no effect; per-variable code detail already appears in each variable's frequency table. The table itself prints only when a filter is active (even one that excluded no cases); otherwise a one-line N statement takes its place. See `?joutput` (`case.processing`).

`print` Logical or NULL. `FALSE` prints nothing – no table, no title, no note – and returns the result as every call does; errors and warnings still show. `TRUE` prints even under `joutput("silent")`. `NULL` (the default) prints unless the level is silent. For a script that keeps many results and wants no output: see `joutput`.

Details

Output is structured consistently with `jdesc()`: a single red "Frequencies" title is printed first, followed by the default-data note (if a juse() default was used), any pipeline messages, and then either the Case Processing table (when a filter is active) or a one-line statement of the N. Each variable then gets its own block: its caption on its own line (the name, or name and label, as `variable.id` directs), a blank line, and the frequency table. Declared missing values get Missing rows of their own below the valid rows. The frequency table ends with a Total row showing the post-pipeline N.

When the table has Missing rows, a "Total valid" row closes the valid rows – the valid N, the base of every Valid % – and, when there is more than one Missing row, a "Total missing" row closes them, as in SPSS FREQUENCIES.

A value that has a value label but no case gets a row of its own at 0, as an empty level of a factor does: a valid value, a declared missing code, and a labelled code inside a declared missing-value range when the range's values are listed one by one. A range with no case in it keeps its single "range" row. When more than ten labelled values (or factor levels) in one table have no case, none of them is listed, and a line under the table says how many there are.

In a text variable, cells with no text – empty, or holding only spaces or tabs – are tabulated together in one row labeled `<blank>`. They are counted as valid values, not as missing: a blank is absent data in one file and a real answer in the next (a column coded "Y" or left empty), and only you know which. A short footnote under the table says how many there are; `jencode` gives them a code,

and jdeclare_missing with codes = "" declares them missing. <blank> is the row's label, not a value in the data: a filter selects those cells by what they hold, subset = trimws(Source) == "" (or Source == "" for the empty cells alone, leaving out those holding only spaces).

A list column (a tibble's, or the geometry column of a spatial data frame), a raw column, and a column that is itself a data frame cannot be tabulated, and are refused before anything prints.

For haven-labelled variables, value labels and numeric codes are combined in the frequency table rows (e.g. 1: Strongly Oppose) at the default value.id setting. Where variable labels are shown, they are shown for all variable types, not only haven-labelled ones.

Given a single variable instead of a data frame, as in jfreq(community$Region), jfreq() tabulates that variable. A column of a data frame is tabulated in its frame, exactly as jfreq(community, Region) tabulates it: the frame's jsubset() and jcomplete() settings and its registrations apply. Any other vector is tabulated on its own. A second variable, and a subset condition naming another variable, need the data frame, and each stops with that form of the call.

With a juse() default set, a name the default data frame has is that frame's variable, as in every jstats function, even when a separate object of the same name exists in the workspace: jfreq(Region) tabulates the default frame's Region, and a line under "Using default data frame" says so. A separate object is read only when the default frame has no variable of that name, or when no default is set.

Value

Invisibly returns a list of class jst_freq containing: frequencies (a named list, one element per variable; each element is itself a list with seven components – valid, the valid-row frequency table as a data frame; udm, the declared-missing rows as a data frame, empty when the variable has none; na, the system-missing row as a one-row data frame, or NULL when there are none; total, valid_count, and missing, the case counts; and var_label, the variable's display label) and sample_info (pipeline and missing data counts).

See Also

jstats for the package overview, workflow conventions, and complete function listing.

Examples

```
# With explicit data frame
jfreq(community, Region)
jfreq(community, Region, Education)

# The data frame may also be given by name
jfreq(data = community, Region)

# Using juse() default
juse(community)
jfreq(Region)
jfreq(Region, Education)

# With a vector directly
jfreq(community$Region)
```

`jlikert` *Register variables as Likert (ordered response) items*

Description

`jlikert()` declares one or more value-labelled variables as Likert items – ordered response scales (for example 1 = Strongly disagree through 5 = Strongly agree). It is the ordered-scale counterpart to `jdummy` (categorical), `jnumeric` (continuous), and `jcount` (count): a variable carries exactly one registered intent at a time, so registering it as Likert clears any prior numeric, count, or dummy registration on it.

Usage

```
jlikert(data, ..., remove = FALSE, clear.all = FALSE)
```

Arguments

`data`	A data frame, or omitted to use the `juse` default. Registrations are stored under the data frame's name, so it is given as a name (or as a place such as `lst$d`); an expression that builds one is refused, with the two lines that name it first. The `NULL` and `remove = TRUE` forms work on a name whose data frame has since been removed.
`...`	One or more unquoted variable names to register, or a single `NULL` to clear this frame's Likert registrations (see Details). The registration applies to numeric variables and to haven-labelled variables that hold numbers. Any other type is refused, and none of the call's variables is then registered: convert a factor or a text variable to numbers first with `jencode`.
`remove`	Logical; if TRUE, remove the named variables' Likert registrations instead of adding them.
`clear.all`	Logical; if TRUE, clear Likert registrations on every data frame.

Details

Scope – display only. The Likert intent refines reporting, not analysis. It sets the variable's subclass to "Likert" in `jscreen`'s Variable Types table, marking it as an ordered scale rather than a generic N-category variable. It does NOT change how any analysis treats the variable (there is no order-aware modelling), and it does not by itself change `jplot` output – a value-labelled small-range variable already plots as an ordered, labelled bar regardless of this registration.

Like the other registration verbs, registrations are session-scoped and keyed by data-frame name; save the frame in R format (.rds) with `jsave` to keep them across sessions.

Clearing mirrors the other registration verbs:

- `jlikert(data, NULL)` – clear this frame's Likert registrations.
- `jlikert(NULL)` – clear the `juse()` default frame (or the sole frame carrying Likert registrations; if several do, it asks rather than clearing them all).

- `jlikert(clear.all = TRUE)` – clear every frame.

`jlikert()` with no arguments prints the current registration status.

Value

Invisibly NULL. Called for its side effect on the session registry.

See Also

`jnumeric`, `jcount`, `jdummy`, `jscreen`

Examples

```
jlikert(community, Environment1, Environment2)  # declare two Likert items
jlikert(data = community, Environment1)         # the data frame by name
juse(community); jlikert(Environment1); juse(NULL)  # or the juse() default
jscreen(community)                              # Sub-class shows "Likert"
jlikert(community, Environment1, remove = TRUE) # undo one
jlikert(community, NULL)                        # clear the registrations
```

---

jlm *SPSS-like linear regression output with standardized coefficients*

Description

Fits a linear model using `stats::lm()` and prints SPSS-style output, including unstandardized coefficients, standard errors, t values, p values, and standardized coefficients (beta). Standardized coefficients are left blank for the intercept and for dummy-coded categorical terms.

Usage

```
jlm(
  formula,
  data,
  subset = NULL,
  variable.id = NULL,
  numeric = NULL,
  categorical = NULL,
  count = NULL,
  ci = NULL,
  std = "regular",
  diagnostics = NULL,
  ref.categories = NULL,
  full = FALSE,
  case.processing.detail = NULL,
  digits = NULL,
  ...,
```

```
  value.id = NULL,
  model.layout = NULL,
  print = NULL
)
```

Arguments

formula	A model formula, e.g. `y ~ x1 + x2`. Transformed terms such as `log(y)` or `I(x1^2)` are computed automatically and used throughout the output.
data	A data frame containing variables referenced in `formula`.
subset	An optional unquoted logical expression (e.g. `Group == 1`) to subset cases for this call only. Applied after jcomplete and jsubset. Does not affect other function calls. Written in R syntax and checked the way `jsubset()` checks a filter: `subset = NOT(Age < 40)`, for example, is refused with the corrected `subset = !(Age < 40)` shown (see `jsubset` for a translation table).
variable.id	Character or NULL. Variable label display mode: one of `"both"`, `"names"`, `"labels"`, `"legend"`, or `"legend.bottom"`. `"names"` shows variable names only; `"both"` shows `"name: label"`; `"labels"` replaces each coefficient's variable name with its label in the Coefficients table (factor level decoration is preserved) – best for short labels; `"legend"` prints a label legend between the Coefficients table and the R-squared/fit block; `"legend.bottom"` prints it at the very end. NULL (default) defers to `joutput()`'s `variable.id` setting. Not a logical.
numeric	Optional. Variables that should be treated as continuous (numeric) even if they have value labels, named bare or in quotes, several with `c()`. For example, `numeric = Age` or `numeric = c(Age, Education)`.
categorical	Optional. Variables that should be treated as categorical even if they lack value labels, named bare or in quotes, several with `c()`. For example, `categorical = Program` or `categorical = c(Program, Region)`. The first sorted unique value becomes the reference category. Use `jdummy()` for control over the reference category.
count	Optional. Variables to treat as counts for this call (the per-call counterpart of `jcount()`), named bare or in quotes, several with `c()`. On the dependent variable it speaks the count-regression caveat definitively rather than as a hedge, and applies even when the variable sits outside the structural 0-6 band. On an independent variable it behaves like `numeric` (a count predictor enters the model as numeric). A variable cannot be listed in both `count` and `categorical`.
ci	Logical or NULL. If TRUE, appends a 95% confidence interval for each unstandardized coefficient (b) at the right of the coefficient table. If NULL (default), defers to `joutput()`'s regression.ci setting (off at minimal and standard, on at full). Computed as the closed form b +/- t(.975, residual df) * SE.
std	Character. Controls the standardized-coefficient column. One of `"regular"` (default) – standardized betas with the prevalence-scaled betas of dummy and dichotomous predictors suppressed, since a fully standardized beta on a 0/1 indicator is not comparable to the continuous betas (an interaction row built on such a predictor is suppressed with it); `"all"` – the same standardized betas

with nothing suppressed; "gelman" – Gelman (2008) scaling (shown for all predictors, and headed "Gelman beta"); "product" – each predictor standardized as it stands, the product of an interaction and a squared term included, as some other software does (shown for all predictors, and headed "Product beta"); or "none" – omit the column. The returned object always carries the full regular betas (beta), the full Gelman betas (beta_gelman) and the full product betas (beta_product) regardless of this display choice.

The regular betas come from refitting the model on z-scored variables, so in a model with an interaction the product is formed from the standardized predictors (Aiken and West, 1991). Some software standardizes the product column itself instead, which is what std = "product" does; see the section Comparing with other software.

The Gelman betas come from refitting the model with every predictor centered, a continuous predictor also divided by two standard deviations and a binary predictor (a 0/1 variable, or a category's dummy variable) left on its one-unit scale; the outcome keeps its own units. A continuous predictor's Gelman beta is the change in the outcome for a move from one standard deviation below its mean to one above; a binary predictor's is the difference between its two groups; the two are comparable, which is the point of the scaling. Every 0/1 column is centered, including the dummy variables of a categorical predictor with three or more categories, which arm::standardize() leaves at 0/1 (its rescaling reaches only the variables named in the formula), so on such a model a main effect that interacts with that predictor is the sample-average effect here and the reference-category effect there.

A computed term that is a power or a product of variables, such as I(x^2) or I(x * z), is recomputed from the rescaled variables in both refits, as Gelman (2008, section 3.1) does: the square of the rescaled x, not the rescaled square. Any other computed term – log(x), a ratio, a condition such as I(x > 10) – is an input of its own and is rescaled as a column.

In a model with an interaction or a squared term, both refits center the predictors, so a main effect's standardized beta is its effect at the average of the predictor it interacts with (a squared term's variable, at its own average), while its b is the effect when that predictor is 0; the two can differ in sign. A note under the coefficient table says so, and points to the section Comparing with other software, whenever such a model shows a standardized column. A variable that equals the product of two other predictors in the model, or the square of one, is named in a note of its own, and under std = "product" a note says which convention the column follows.

diagnostics Logical, character vector, or NULL. If TRUE, prints the VIF table (for a model with two or more predictors) and draws the five diagnostic plots. A character vector names the ones to show: vif, residuals, qq, scale, cooks, leverage, more than one combined with c(), as in diagnostics = c("vif", "qq"); a name that is not one of these is an error. A VIF above 10 gets a line of interpretation under the table, except at joutput("minimal"). In a model with a predictor of three or more categories, whose dummy variables would each get a VIF that depends on the reference category, the table gives each predictor one row with its generalized VIF (Fox and Monette, 1992), Df and GVIF^(1/(2*Df)), *as* car::vif() *does, the dummy variables sharing a row (and an interaction's,*

*with the same other variable); squared, GVIF^(1/(2*Df)) is on the scale of a VIF, and the line of interpretation fires when it passes 10. If NULL (default), defers to `joutput()`'s `diagnostics` setting, which is off at every output level until it is set. The same plots are available afterwards from `jplot()` on the returned object.

`ref.categories` Logical or NULL. Per-call override for showing the reference-categories block (the baseline level dropped from each set of dummy variables). `NULL` (default) defers to `joutput()`'s `ref.categories` setting. Applies to `jlm()` and `jlogistic()` only, since they are the functions that produce dummy-coded coefficient tables.

`full` Logical. If TRUE, turns on the coefficient confidence interval. Does not override an explicit FALSE, and does not turn on diagnostics, which `diagnostics` alone governs.

`case.processing.detail`
Per-call override of the Case Processing Summary detail tier: one of `"none"`, `"totals"`, or `"per_code"`. `NULL` (default) uses the active `joutput()` level default. The Case Processing table itself prints only when a filter or listwise deletion excluded cases; otherwise a one-line N statement takes its place. See `?joutput` (`case.processing`).

`digits` Integer or NULL. Number of decimal places for continuous statistics in the output tables (range 0-7; `digits = 0` prints whole numbers with no trailing decimal point). Does not affect p-values, percentages, or integer quantities (counts, N, degrees of freedom), which keep their own fixed conventions. NULL (default) defers to `joutput()`'s `digits` setting (default 3).

`...` Reserved for argument-name checking. Passing `which`, `plots`, or `show` will produce a helpful error suggesting `diagnostics` instead.

`value.id` Character or NULL. Value-label display mode for the dummy category rows in the Coefficients table: one of `"both"` (`"code: label"`, degrading to a bare code where a code has no label), `"values"` (the bare code), or `"labels"` (the value label, degrading to the bare code where none exists). The reference category folded into each grouped variable's header follows the same mode. `"legend"` and `"legend.bottom"` are not supported here: a coefficient table already pairs each value label with its row, so a separate legend block would only duplicate it. Passing either explicitly is an error; a `joutput()` default of `"legend"` or `"legend.bottom"` is tolerated and rendered as `"both"`, so it does not break a bare call. Variables with no value labels render identically under all supported modes. NULL (default) defers to `joutput()`'s `value.id` setting. Applies only to multi-category dummy predictors; continuous and single-contrast (dichotomous) predictors are unaffected. Not a logical.

`model.layout` Character or NULL. The order of the coefficient table and the fit statistics: `"coefficients"` (the default) prints the Coefficients table first and the fit block after it; `"fit"` prints the fit block first, for readers who look at the model as a whole before its coefficients. The Case Processing block, the diagnostics and the other tables stay where they are. NULL (default) defers to `joutput()`'s `model.layout` setting.

`print` Logical or NULL. `FALSE` prints nothing – no table, no title, no note – and returns the result as every call does; errors and warnings still show. `TRUE` prints even

under `joutput("silent")`. `NULL` (the default) prints unless the level is silent. For a script that keeps many results and wants no output: see `joutput`.

Details

Also prints key model summary information (R-squared, adjusted R-squared, residual standard error, F-test, sums of squares, and N). If a predictor is dropped because it is an exact linear combination of the others (perfect collinearity), a warning names it.

A model with no predictors, `y ~ 1`, is fitted too: its one coefficient is the mean of the outcome, tested against 0, R-squared is 0, and the F line reads "none (the model has no predictors to test)".

A red "Linear Regression" title is printed first, followed by variable labels (if present), then the coefficient table and model fit statistics.

Handling of variables:

- Variables registered with `jdummy()` are expanded into dummy variables using the registered reference category. A registered category with no case in the analysis sample (removed by a filter, or emptied by listwise deletion) is left out of that model; if it is the reference category, the first remaining category is the reference for that model instead. A note reports either, and the coefficient table's header names the reference actually used. The registration itself is not changed.
- Unregistered haven-labelled variables with value labels are automatically treated as categorical (converted to factors). The first category is used as the reference, and an informational message suggests using `jdummy()` for control over the reference category.
- Haven-labelled variables without value labels are treated as continuous (converted to numeric).
- The `numeric` argument overrides auto-detection for variables that have value labels but should be treated as continuous (e.g. Age with labels like "18 years", "19 years").
- The `categorical` argument forces variables without value labels (or plain numeric variables) to be treated as categorical (e.g. a numeric Program variable coded 1–4 from a CSV file).
- The dependent variable is always modelled as numeric. Naming it in `numeric` or `count` does not change that; it only asserts the DV's role so the count / categorical-like note is silenced (`numeric`) or stated definitively (`count`).

Transformed terms in `formula` are computed automatically. A term that applies a function to a variable – `log(x)`, `sqrt(x)`, `exp(x)`, `I(x^2)`, `scale(x)`, an arithmetic expression, or a logical condition such as `I(x > 10)` – is evaluated once on the analysis data and enters the model as a single derived column named for the expression, so the coefficient table, the group descriptives, and the standardized-beta refit all report the term as written. This follows the base R formula convention; the terms supported inline are those that evaluate to one numeric or logical column. Terms that produce several columns (`poly(x, 2)`, spline bases) or a categorical result (`cut(x, 3)`) are not supported inline: create the derived variable as a column of the data first, then name that column in the formula. A categorical variable may be compared but not transformed. `I(x == 2)`, `I(x %in% c(1, 3))` and, for a text variable, `I(x == "web")` each enter the model as a TRUE/FALSE predictor, as in `lm()`; an order comparison such as `I(x >= 3)` does too when `x` holds numbers. Arithmetic on a categorical `x` – `log(x)`, `I(x^2)` – is refused, because it would be computed from the category codes; register `x` with `jnumeric` if its codes are meant as numbers. A value or vector from your workspace may be named inside a computed term, as in `lm()`: `I(x > cutoff)` with `cutoff <- 10`. A data frame may not: write `y ~ x` with `data = MyData`, not `MyData$y ~ MyData$x`. A variable may

not: jstats analyzes the variables of a data frame, so a vector holding one value for each case – `w` in `I(x * w)`, or an object reached with `$` such as `lst$w` – is added to the data frame first (`MyData$w <- w`), where `lm()` would read it from the workspace; the stop gives that line. A vector of any other length used value by value is refused, where `lm()` would repeat the shorter one; a set used with `%in%` may have any length.

In a formula, `^` applied to a single variable is not arithmetic: it is R's operator for interactions up to that order, so `x^2` enters the model as `x` alone. Write `I(x^2)` for the square. `jlm()` fits the model as written and then warns when a formula does this. A power that is not a number, `x^k`, stops with an error, as in `lm()`; write `I(x^k)`.

Value

Invisibly returns a list of class `jst_lm` containing:

model The fitted `lm` object.

model_type Character string `linear`.

model_frame The model frame used to fit the model.

formula_used The formula after dummy expansion.

coefficients Formatted coefficient table (data frame); includes 95% CI Lower / Upper columns when `ci` is on.

coefficients_raw Flat data frame of raw, full-precision coefficient statistics (one row per coefficient): `term` (machine key), `b`, `SE`, `t`, `df`, `p`, `beta`, `beta_gelman`, `beta_product`, and `ci_lower` / `ci_upper` bounds (present regardless of the `ci` display toggle). Carries `beta_standardization` and `outcome` attributes.

fit_raw List of raw, full-precision fit statistics (R-squared, adjusted R-squared, residual SE, F with its dfs and p-value, residual df, and N).

r_squared R-squared value.

adj_r_squared Adjusted R-squared value.

residual_se Residual standard error.

f_statistic Named numeric vector with F value, df1, df2, and p. For a model with no predictors, F and p are `NA` and df1 is 0.

sums_of_squares Named numeric vector (regression, residual, total).

n Number of observations used in the model.

dummy_coef_names Names of dummy variable columns created by `jdummy()` registrations.

ref_cats Reference category descriptions for all categorical variables in the model.

dummy_groups Named list, one element per dummy-coded variable: `labels`, its categories as the coefficient table labels them, and `dummy`, the dummy column of each (`NA` for the reference). Used by `jplot()`.

vif Named numeric vector of VIF values, or NULL for bivariate.

gvif The generalized VIFs, when the model has a set of dummy variables (a predictor of three or more categories): a data frame of `term`, `GVIF`, `Df` and `GVIF_adj` (`GVIF^(1/(2*Df))`), one row per predictor, as the VIF table prints them; NULL otherwise.

sample_info Pipeline and missing data counts.

Interactions

An interaction is entered in the formula with `*`, as in R: `y ~ x * z` fits `x`, `z` and their product, and the product's row in the coefficient table is labeled `x * z`. With a categorical predictor registered with `jdummy`, the interaction rows are grouped under a header naming the reference category – `Region * Age (ref = 1: North)` – with one row per dummy variable, as the predictor's own rows are. Each row's b is the change in the slope of one variable per unit of the other (or, for a dummy row, the difference in slope between that category and the reference), and the main-effect rows give the slopes where the other variable is zero. A note under the table says that the standardized column is built from centered predictors, which is why a beta there can have the opposite sign from its b, and points to the section Comparing with other software for how the standardized columns are computed. `jplot` on the result draws the effect plots.

Comparing with other software

Two conventions exist for the standardized coefficients of a model with an interaction or a squared term, and they give different numbers for the same model.

The first convention standardizes the inputs and forms the product, or the square, from them. The default column and the Gelman column follow it: `std = "regular"` refits the model on z-scored variables (Aiken and West, 1991), and `std = "gelman"` refits it on centered, rescaled predictors (Gelman, 2008). In R, `arm::standardize()` follows it for the Gelman scaling, and the default refit of `effectsize::standardize_parameters()` for the regular one; both agree with jstats when every predictor is numeric. Both differ from jstats on a categorical predictor of three or more categories. jstats rescales its dummy variables (see `std`); those packages take the predictor as an R factor (base R's form for a categorical variable, where jstats uses a `jdummy()` registration) and leave its dummy variables at 0/1. Some of their betas then differ, among them the beta of a main effect that interacts with that predictor.

The second convention standardizes the product column itself, as though the product were an ordinary predictor, and a squared term's column the same way. SPSS REGRESSION reports this beta, since there the product is computed as a new variable and entered like any other, and `effectsize::standardize_parameters(method = "basic")` gives it in R. In jstats, `std = "product"` gives it. On this convention the betas of the product and of the main effects differ from those of the first, and they depend on where each variable's zero falls: adding a constant to one of the variables fits the same model but changes them, while the first convention's betas stay the same.

Only the standardized columns differ: b, its standard error, t, p and R-squared are the same under both. The same split appears within jstats. A product computed by hand and entered as its own variable (a column `xz` made from `x * z`, then `y ~ x + z + xz`) fits the same model as `y ~ x * z`, but jstats cannot see its parts, so its standardized columns follow the second convention; a note under the table names the variable when it equals the product of two other predictors in the model, or the square of one.

jstats follows the first convention and recommends it: enter an interaction in the formula, as `x * z`, and a squared term as `I(x^2)`. If you prefer the second convention instead – to match SPSS REGRESSION, say – use `std = "product"`.

References

Aiken, L. S., and West, S. G. (1991). *Multiple Regression: Testing and Interpreting Interactions*. Sage.

Gelman, A. (2008). Scaling regression inputs by dividing by two standard deviations. *Statistics in Medicine*, 27(15), 2865-2873.

See Also

jstats for the package overview, workflow conventions, and complete function listing.

Examples

```
# With explicit data frame (named argument)
jlm(WellbeingScore ~ Income + Age, data = community)

# With explicit data frame (positional argument)
jlm(WellbeingScore ~ Income + Age, community)

# Using juse() default
juse(community)
jlm(WellbeingScore ~ Income + Age)

# CATEGORICAL PREDICTORS
#
# Per-call: categorical = ... applies for one call only and does not
# persist. Useful for a quick one-off analysis.
jlm(WellbeingScore ~ Region + Age, categorical = "Region")

# The recommended approach for repeated analyses: register the variable
# with jdummy() before running jlm(). This sets the categorical
# treatment persistently, so subsequent jlm() calls (and other
# analyses) use the same coding without re-specifying.
jdummy(community, Region)
jlm(WellbeingScore ~ Region + Age)

# To choose a non-default reference category:
jdummy(community, Region, ref = "West")
jlm(WellbeingScore ~ Region + Age)

# FORCING NUMERIC TREATMENT
#
# Use numeric = ... when a variable has value labels (haven_labelled)
# but you want it treated as a continuous score (e.g., a Likert
# scale you want the slope-per-unit interpretation for).
jlm(WellbeingScore ~ Age + Education, numeric = "Education")

# Multiple overrides at once
jlm(WellbeingScore ~ Education + Environment4 + Smoker,
    numeric = c("Education", "Environment4"), categorical = "Smoker")

# STANDARDIZED COEFFICIENTS
#
# The default (std = "regular") leaves the beta blank on the dummy rows.
# std = "all" shows it there too; std = "gelman" puts continuous and
# dummy predictors on one comparable scale; std = "none" drops the column.
jlm(WellbeingScore ~ Region + Age, std = "all")
```

```
jlm(WellbeingScore ~ Region + Age, std = "gelman")
jlm(WellbeingScore ~ Region + Age, std = "none")

# An interaction, entered in the formula so the standardized column is
# built from the inputs (see the section Comparing with other software)
jlm(Flourishing ~ SocialSupport * Stress, data = clinic)

# A squared term, written with I(); its standardized value is built from
# the standardized Stress
jlm(Flourishing ~ Stress + I(Stress^2), data = clinic)

# The second convention: each predictor standardized as it stands, the
# beta SPSS REGRESSION reports for a product computed as a new variable
jlm(Flourishing ~ SocialSupport * Stress, data = clinic, std = "product")

# jdummy(community, NULL) clears its registration -- not normally needed.
# You'd clear a default or registration only to undo a mistake, or -- as
# in this example -- to reset state for testing.
jdummy(community, NULL)
juse(NULL)
```

`jload` *Load a data file into R*

Description

`jload()` reads a data file and assigns it as a data frame in your environment. Supports SPSS (`.sav`), Stata (`.dta`), SAS (`.sas7bdat`, `.xpt`), Excel (`.xlsx`, `.xls`), CSV (`.csv`), and R's native `.rds` format.

The file format is determined entirely by the file extension — `jload()` reads the extension (e.g. `.sav`, `.dta`, `.xlsx`) and uses the appropriate reader automatically.

By default, `jload()` looks for the file in the working directory. If a data folder is configured with `joptions(data.dir = ...)`, that folder is searched first. If a full file path is provided, it is used directly.

The data frame is automatically named after the file (without the extension). Use the `name` argument to specify a different name.

Usage

```
jload(
  file,
  name = NULL,
  use = FALSE,
  overwrite = FALSE,
  package = FALSE,
  check.missing = TRUE,
  sheet = NULL,
```

```
  skip = 0,
  header = TRUE,
  text = NULL,
  na = NULL,
  preserve.declarations = TRUE,
  missing.notice = NULL,
  quiet = FALSE,
  ...
)
```

Arguments

file	Character string. The filename (e.g. "mydata.sav") or a full file path (e.g. "C:/Projects/mydata.sav"). Use forward slashes in file paths. If the extension is omitted, jload() searches for common data file types automatically.
name	Character string (optional). The name to assign the data frame in your environment. If omitted, the name is derived from the filename.
use	Logical. If TRUE, automatically calls juse() on the loaded data frame to set it as the default for jstats functions. Default is FALSE.
overwrite	Logical. If TRUE, overwrites an existing object with the same name without prompting. If FALSE (default), prompts for confirmation in interactive sessions. In non-interactive sessions, overwrites and says so in a note. In a script, state it explicitly: jload("mydata.rds", overwrite = TRUE). Otherwise, if the name already exists in your environment, and the script was run by pasting or with RStudio's Run button, the call stops with an error.
package	Logical. If TRUE, loads a jstats example dataset shipped in the package (e.g. community, clinic) by bare name, or a shipped example file by its file name ("community_spss.sav"), bypassing the disk search. Use this when a same-named file in the working directory or data folder would otherwise shadow the shipped copy. If FALSE (default), a matching disk file takes precedence and the shipped copy is used only when no file matches. file must be a name with no path when package = TRUE, and a dataset's name has no extension.
check.missing	Logical. If TRUE (default), scans numeric variables for values that look like coded missing values (e.g. -99, 999) and reports them. Set to FALSE to skip this check.
sheet	For Excel files only. The sheet to read — either a sheet name (character) or sheet number (integer). Defaults to the first sheet. If the file has multiple sheets and sheet is not specified, a message lists the available sheets.
skip	For Excel and CSV files. The number of rows of the sheet, or lines of the file, above the variable names: a title, a blank row, a note. Default 0. When the first row of a sheet looks like a title, a note shows the call with the skip it needs; a CSV file in that shape stops with the same call.
header	For Excel and CSV files. TRUE (default) reads the first row of the sheet, or line of the file, as the variable names. FALSE reads it as the first case, for a file with no names row, and names the variables V1, V2, and so on, as base R's read.csv(header = FALSE) does. Every other format carries its own variable

names, so header = FALSE stops there. When every name read is a number, a note shows the call with header = FALSE; writing header = TRUE in the call says the numbers are the names (years, say), and the note is not shown.

text — For Excel and CSV files. Variables to read exactly as the file has them, as text: no number, no TRUE or FALSE, the word NA kept, and na not applied. Name them without quotes, or in quotes, alone or in c(): text = Diag, text = c(Diag, Postcode). A name is the file's own or the one jload() gives the variable ("Post code" or Post_code). For codes that numbers would change, such as the diagnosis codes 250.10 and 250.1. A name a CSV file does not have stops the load before it is read, listing the file's names (a sheet's, once it is read). Every other format carries its own variable types, so text stops there.

na — For Excel and CSV files. Cell values to read as missing, in quotes: na = c("N/A", "-", "."). They are added to what is already missing (an empty cell in a variable of numbers, and in a CSV file the word NA there), in every variable but those named in text. When a variable is numbers except for a few cells, a note names those cells and shows the call with na. Every other format carries its own missing values, so na stops there.

preserve.declarations — Logical. If TRUE (default), declared missing values arriving with the file are preserved: SPSS-style codes such as -99, and declared ranges, keep their original numeric values in the data frame, with the declaration attached so the package's analysis functions still treat them as missing, and Stata-style tagged values (.a, .b, ...) are kept as read. If FALSE, both forms are converted to plain NA on import and the declarations are removed; value labels are kept. Applies to any loaded file whose columns carry missing-value declarations — typically .sav, .dta, and .sas7bdat files, and .rds files saved from such data. For .sav files, TRUE corresponds to haven's user_na = TRUE. The haven package and its documentation call these user-defined missing values.

missing.notice — Per-call override for the missing-value notification. NULL (default) defers to the setting from joutput(). TRUE prints the notification, in its full form, on any load with declared missing values; FALSE suppresses it. Under the default (standard) and full output levels the first such load in a session prints the full notification and later loads print a compact form (the variable inventory, plus the convention note when one applies, without the guidance lines); minimal suppresses the notification. See ?joutput for the full toggle behavior.

quiet — Logical; default FALSE. When TRUE, suppresses jload()'s informational messages (the directory-resolution note, file found, load summary, default-data note, registration notes, and the narrative about declared missing values, overriding missing.notice). Errors, warnings, the multi-sheet advisory, the overwrite prompt, the note that a local file shadowed a shipped dataset, the note that an existing object was replaced, and the notes on what the read changed or found (variable names changed, a title row, names that are all numbers, rows and columns that may not be data, a CSV file's encoding, a variable kept as text, a variable of numbers made text by a few cells) are still shown, as is the coded-missing scan report; turn that off with check.missing = FALSE.

... — Not used. An input jload() does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Details

File paths: Use forward slashes (/) in file paths. If you copy a path from Windows File Explorer, replace the backslashes with forward slashes. R does not accept single backslashes in file paths: inside quoted text it reads a backslash and the letter after it as one special character (a backslash before t is a tab), so the path that arrives is not the one typed. `jload()` stops on such a path and shows it written with forward slashes.

CSV files: A CSV file with one column keeps its empty lines: each is a case whose one cell is empty, which is how a missing value is written there. In a file with two or more columns an empty line is not a case. An empty cell in a text variable is read as blank text, never as missing (see `jsave`).

Before a CSV file is read, every line's values are counted. A first line with fewer values than the lines below it (a title above the names, or a name missing), and a later line with more values than the first, stop the load with the line and what to do: read as they are, R would turn the first values of every line into row names, or split the long line into two cases, and say nothing. A file whose values are separated by semicolons, as Excel saves them in many European settings, stops with the base R line that reads it. A file that is not in UTF-8 (one saved by Excel on Windows as "CSV (Comma delimited)" is in that computer's own encoding, Windows-1252 for English and Western European languages) is read in Windows-1252, and a note says so and how to save the file in UTF-8. Read with `header = FALSE`, a file has no names line, so each line is measured against the number of values most lines hold: a line holding more stops the load, and one holding fewer is read with its last cells empty.

Values in a CSV file: A CSV file stores every value as text, so each variable's type is read from its values. A variable becomes numbers, or `TRUE` and `FALSE`, only where no value would change:

- A whole number written with a leading zero (a postcode `0800`, an ID `007`) keeps its variable as text, as written; numbers would drop the zero. A note says so, with the line that makes it numbers anyway (and, for a variable `jload()` renamed, the line that puts its label back).
- A whole number of more than 15 digits (an account or record ID) keeps its variable as text: a number holds 15 digits exactly, so different IDs would become one number. A note says so.
- `T` and `F` standing alone stay text (a variable of `F` in a sample of women is not `FALSE`); the words `TRUE` and `FALSE`, as R and Excel write them, are read as `TRUE` and `FALSE`.
- The word `NA` is missing in a variable of numbers, or of `TRUE` and `FALSE`, which is how R writes a missing value there. In a text variable it is a word, as in an Excel sheet (`NA` is Namibia's country code), and the coded-missing scan names it with the line that declares it missing.

Base R's `read.csv()` reads all four as numbers, `TRUE` or `FALSE`, or missing, without a word. Use `text` for a code the rules cannot see, such as `250.10`, which numbers would read as `250.1`.

Values that are not numbers: In a CSV file or a sheet, a variable of numbers that holds a few other values – `N/A`, `-` or `.` for a missing value – is read as text. jstats does not assume such a value is missing (a dash can mean zero), so a note names the values, with the `jload()` call whose `na` reads them as missing: `jload("incomes.csv", overwrite = TRUE, na = c("N/A", "-"))`. A variable is named when it has at most five such values and numbers in more than half of its cells that are not empty; in a sheet, number cells must be more than half of them. A sheet's variable that also holds a number stored as text stays text whatever `na` names, so its note gives the base R lines that make it numbers instead: `d$Code[d$Code == "N/A"] <- NA`, then `d$Code <- as.numeric(d$Code)`. Where `jload()` renamed the variable, a last line puts back the label it gave it, which `as.numeric()` drops.

A file with no names row: A CSV file or a sheet whose first row is data loses that case to the variable names, which are then numbers (`1` becomes `X1`). When every name read is a number, or empty, a note says the row they came from may be the first case and shows the call that reads it as one: `jload("scores.csv", header = FALSE)`, the variables named `V1`, `V2`, and so on.

Rows and columns that may not be data: A sheet can hold rows with nothing in them, a column with no name and nothing in it, and lines of text below the data, such as a source or a footnote; a CSV file can hold a line of commas alone, an empty column from a comma at the end of each line, and the row numbers R's `write.csv()` writes in a first column with no name. Each is read as data, as a case or a variable, and a note names them by their place in the file, with the case numbers or the variable names; when footnotes turned a variable of numbers into text, it says so. Nothing is removed. An empty line in a CSV file of one column is a case with a missing value, and is not named.

Variable names from other software: A file from outside R can hold names R cannot use as they are: `2024 score`, `Age (years)`, an empty header, two columns with the same name, a word R reserves (`if`, `TRUE`), an SPSS name such as `Q1@A`, a Stata name such as `_merge`. For every format but `.rds`, each such name is changed: spaces and symbols at either end are dropped, any other run of characters that are not letters, digits or underscores becomes one underscore, a name that starts with a digit or an underscore or is a reserved word gets an `X` in front, an empty name becomes `V` and its column's position, a repeated name gets `_2`, and a changed name is kept to 32 characters. So `2024 score` becomes `X2024_score` and `Age (years)` becomes `Age_years`. A name R can use as it is is never changed. The original name becomes the variable's label, unless the variable has a label of its own, and a note lists every change. An `.rds` file is returned as it was saved.

File search order:

1. If the path contains a directory separator (`/`, or a backslash), the path is used directly.
2. If the path is a bare filename, `jload()` checks: (a) the folder named by the `data.dir` setting in `joptions` if it is set and exists; (b) the working directory.

When the extension is omitted, both locations are searched for every supported extension, and more than one match (two extensions, or the same file in both locations) is an error that asks for the extension. If nothing matches on disk, a bare name falls back to a shipped example dataset of that name (see below).

Auto-naming: The data frame name is derived from the filename by stripping the extension. If the resulting name starts with a digit (which R does not allow as a variable name), you must supply the `name` argument; the error offers one (`2024.csv` is offered `name = "X2024"`). Other characters R does not allow in a name, such as spaces and hyphens, are changed as variable names are (see above): `"my data-1.sav"` loads as `my_data_1`.

Excel files: Excel files (`.xlsx`, `.xls`) do not contain variable or value labels. The data will be loaded as plain numeric, character, or logical columns; a variable whose name was changed has its original name as its label. Use `jrelabel()` to add labels after loading if needed. Row 1 of the sheet is read as the variable names; when the names are lower down, `skip` gives the number of rows above them, and when there are none, `header = FALSE` reads row 1 as data. Each variable's type is read from every row of the sheet. In a text variable, an empty cell and a cell holding only spaces are read as blank text, as in a CSV file: one category, `<blank>`, counted as a valid value and never assumed missing. To treat them as missing, declare them with `jdeclare_missing()` and `codes = ""`. An empty cell in a variable of numbers, or of numbers stored as text, is missing. With `na = ""` every empty cell is read as missing.

Missing-value declarations: Missing-value declarations are stored in the file itself, but they only survive the trip back if the reader requests them. `jload()` always does, so declarations written by `jsave()` are present after every jstats load (unless `preserve.declarations = FALSE`). Other ways of reading the same file may convert the declared cells to plain `NA` and discard the declarations, so the same file can show different numbers of valid cases depending on how it was read.

Coded missing values: When `check.missing = TRUE`, the function scans numeric variables for values that look like coded missing values but are not declared as missing. (Declared missing values are reported in the load's own notification, never by the scan.) Only whole-number values are considered. A value is flagged when it is

- a negative number at least three times the size of the variable's largest non-negative value (-99 on a 1-to-5 scale), or
- at least five times the size of every other value in the variable (999 on a 1-to-7 scale).

A value in the `missing.convention.codes` setting of `joptions` (-99, -98 and -97 by default) is flagged as well, whatever its size. A flagged value whose value label suggests missingness (such as "Refused" or "Don't know") is reported as label-only; one of the setting's codes, as convention-matched; the rest as suspected. Flagged values are reported but not changed. If they are missing values, declare them with `jdeclare_missing`, which keeps the codes and their labels; the report shows the call.

A text variable's values are flagged too: as label-only when their value label suggests missingness, and as suspected when the value itself does (`"UNKNOWN"`, `"Not stated"`) or when it differs from a value the variable declares only in letter case or outer spaces. The call the report shows names the values the variable already declares, since a declaration replaces them, and at most three in all.

Package example datasets and .rda / .RData files: `jload()` opens the example datasets shipped with jstats – currently `community` and `clinic` – by bare name: `jload("community")`. These ship inside the package as .rda files, but the bare-name load is a lookup of the shipped dataset, not a file read, and it works only for package datasets. It does not extend to .rda or .RData files generally: naming the extension (`jload("community.rda")`) or pointing at any other .rda or .RData file is refused with a pointer to base R's `load()`, because such files can hold several objects under names of their own choosing, outside jload's one-data-frame contract. Note also that the shipped dataset is a fallback: a file with a matching name in the working directory or data.dir folder is opened instead of the shipped copy. Use `package = TRUE` to force the shipped dataset.

Example files: jstats also ships the `community` data as files, the kind a collaborator might send: `community_spss.sav`, `community_stata.dta` and `community_excel.xlsx`. When no file of that name is in the data.dir folder or the working directory, `jload("community_spss.sav")` reads the shipped one, and says so; the name without its extension works too (`jload("community_spss")`). A file of the same name on disk is read instead, with a note giving the call that reads the shipped file: `jload("community_spss.sav", package = TRUE)`.

Variables named T or F: At `joutput("full")` a note names a variable called `T` or `F`. jstats reads it as the variable, but in base R `T` and `F` also stand for `TRUE` and `FALSE`, and code run inside the data frame – `with()`, `subset()` – reads the variable in their place.

Value

Invisibly returns `NULL`; jload() is called for its side effects. The loaded data frame is placed in the calling environment under the file's name (or `name`), and any classification registrations saved

with an .rds file are restored for that name. Loading a file that carries none clears any registrations set this session under that name, with a note saying so. Do not assign the result: `x <- jload("mydata.rds")` binds only `NULL`, while the data frame still arrives under its own name.

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
## Not run:
# SPSS
jload("community.sav")
jload("community.sav", use = TRUE)
jload("community.sav", name = "MySurvey")

# Stata
jload("community.dta")

# SAS
jload("community.sas7bdat")
jload("community.xpt")

# Excel
jload("community.xlsx")
jload("community.xlsx", sheet = "Wave2")
jload("community.xlsx", sheet = 2)
jload("survey.xlsx", skip = 2)   # a title and a blank row above the names
jload("scores.csv", header = FALSE)   # no names row: V1, V2, ...
jload("visits.csv", text = Diag)       # codes kept as written: 250.10
jload("incomes.csv", na = c("N/A", "-"))   # these cells read as missing

# CSV and R format
jload("community.csv")
jload("community.rds")

# Extension omitted -- jload searches for a matching file automatically
jload("community")

# The example files shipped with jstats, read by name
jload("community_spss.sav")
jload("community_excel.xlsx", package = TRUE)   # even beside a local copy

# Full file path
jload("C:/Projects/Data/community.dta")

# Quiet load (e.g. in a .Rprofile or startup script): suppresses the
# informational messages while still loading. Errors and warnings still show.
jload("community.rds", name = "MyData", quiet = TRUE)

## End(Not run)
```

`jlogistic` *Logistic regression with SPSS-style output*

Description

Fits a binary logistic regression using `stats::glm()` with `family = binomial` and prints formatted output including an omnibus model test, model summary statistics, and a coefficients table with Gelman-standardized coefficients and odds ratios (Exp(b)).

Usage

```
jlogistic(
  formula,
  data,
  subset = NULL,
  variable.id = NULL,
  numeric = NULL,
  categorical = NULL,
  count = NULL,
  ci = NULL,
  classification = FALSE,
  diagnostics = NULL,
  ref.categories = NULL,
  full = FALSE,
  case.processing.detail = NULL,
  digits = NULL,
  ...,
  value.id = NULL,
  model.layout = NULL,
  std = "gelman",
  print = NULL
)
```

Arguments

`formula`	A model formula, e.g. `DV ~ IV1 + IV2`. The DV must be a binary variable coded 0/1, and a variable of the data: a computed term on the left, such as `I(Score > 50)`, is refused, so create the 0/1 variable first. Transformed predictor terms such as `log(IV1)` are computed automatically and used throughout the output.
`data`	A data frame containing variables referenced in `formula`.
`subset`	An optional unquoted logical expression (e.g. `Group == 1`) to subset cases for this call only. Written in R syntax and checked the way `jsubset()` checks a filter: `subset = NOT(Age < 40)`, for example, is refused with the corrected `subset = !(Age < 40)` shown (see `jsubset` for a translation table).
`variable.id`	Character or NULL. Variable label display mode: one of `"both"`, `"names"`, `"labels"`, `"legend"`, or `"legend.bottom"`. `"names"` shows variable names

only; "both" shows "name: label"; "labels" replaces each coefficient's variable name with its label in the Coefficients table (factor level decoration is preserved) – best for short labels; "legend" prints a label legend just below the Coefficients table (at the coefficients/fit seam); "legend.bottom" prints it at the very end. NULL (default) defers to joutput()'s variable.id setting. Not a logical.

numeric Optional. Variables to treat as continuous even if they have value labels, named bare or in quotes, several with c().

categorical Optional. Variables to treat as categorical even if they lack value labels, named bare or in quotes, several with c(): categorical = Region.

count Optional. Independent variables to treat as counts for this call (the per-call counterpart of jcount()), named bare or in quotes, several with c(). A count predictor is numeric-like, so it enters the model exactly as numeric would; the argument is provided for symmetry with the other analysis functions. The binary dependent variable is fixed, so naming it here has no effect. A variable cannot be listed in both count and categorical.

ci Logical or NULL. If TRUE, adds 95% confidence intervals for Exp(b). If NULL (default), defers to joutput().

classification Logical. If TRUE, prints a classification table showing predicted vs observed outcomes. Default is FALSE.

diagnostics Logical, character vector, or NULL. If TRUE, prints the VIF table (for a model with two or more predictors) and draws three diagnostic plots: binned residuals, Cook's distance, and residuals against leverage. A character vector names the ones to show: vif, binned, cooks, leverage, more than one combined with c(), as in diagnostics = c("vif", "cooks"); a name that is not one of these is an error. The VIF values are the predictors' ordinary variance inflation factors, computed from their correlations as for a linear model. A VIF above 10 gets a line of interpretation under the table, except at joutput("minimal"). In a model with a predictor of three or more categories the table gives each predictor one row with its generalized VIF, Df and GVIF^(1/(2*Df)), the dummy variables sharing a row, as in jlm(); computed from the correlations, these differ a little from car::vif() on a glm, which uses the coefficients' covariance. If NULL (default), defers to joutput()'s diagnostics setting, which is off at every output level until it is set. The plots, and the probability, ROC, calibration and odds-ratio plots, are available afterwards from jplot() on the returned object.

ref.categories Logical or NULL. Per-call override for showing the reference-categories block (the baseline level dropped from each set of dummy variables). NULL (default) defers to joutput()'s ref.categories setting. Applies to jlm() and jlogistic() only, since they are the functions that produce dummy-coded coefficient tables.

full Logical. If TRUE, turns on ci and classification. Does not override an explicit FALSE, and does not turn on diagnostics, which diagnostics alone governs.

case.processing.detail
Per-call override of the Case Processing Summary detail tier: one of "none", "totals", or "per_code". NULL (default) uses the active joutput() level

default. The Case Processing table itself prints only when a filter or listwise deletion excluded cases; otherwise a one-line N statement takes its place. See `?joutput` (`case.processing`).

`digits` Integer or NULL. Number of decimal places for continuous statistics in the output tables (range 0-7; `digits = 0` prints whole numbers with no trailing decimal point). Does not affect p-values, percentages, or integer quantities (counts, N, degrees of freedom), which keep their own fixed conventions. NULL (default) defers to `joutput()`'s `digits` setting (default 3).

`...` Reserved for argument-name checking. Passing `which`, `plots`, or `show` will produce a helpful error suggesting `diagnostics` instead.

`value.id` Character or NULL. Value-label display mode for the dummy category rows in the Coefficients table: one of `"both"` (`"code: label"`, degrading to a bare code where a code has no label), `"values"` (the bare code), or `"labels"` (the value label, degrading to the bare code where none exists). The reference category folded into each grouped variable's header follows the same mode. `"legend"` and `"legend.bottom"` are not supported here: a coefficient table already pairs each value label with its row, so a separate legend block would only duplicate it. Passing either explicitly is an error; a `joutput()` default of `"legend"` or `"legend.bottom"` is tolerated and rendered as `"both"`, so it does not break a bare call. Variables with no value labels render identically under all supported modes. NULL (default) defers to `joutput()`'s `value.id` setting. Applies only to multi-category dummy predictors; continuous and single-contrast (dichotomous) predictors are unaffected. Not a logical.

`model.layout` Character or NULL. The order of the coefficient table and the fit statistics: `"coefficients"` (the default) prints the Coefficients table first and the fit block after it; `"fit"` prints the fit block first, for readers who look at the model as a whole before its coefficients. The Case Processing block, the diagnostics and the other tables stay where they are. NULL (default) defers to `joutput()`'s `model.layout` setting.

`std` Character. The standardized-coefficient column, between p and Exp(b). One of `"gelman"` (default) – Gelman (2008) scaling, headed "Gelman beta"; `"product"` – each column of the model scaled as it stands, the product of an interaction and a squared term included (headed "Product beta"); or `"none"` – omit the column. See the section Standardized coefficients. `jlm()`'s regular standardization (`"regular"`, `"all"`) divides by the outcome's standard deviation, which a binary outcome does not have, and is not available here. The returned object carries both columns' full values (`beta_gelman`, `beta_product`) whichever is shown.

`print` Logical or NULL. `FALSE` prints nothing – no table, no title, no note – and returns the result as every call does; errors and warnings still show. `TRUE` prints even under `joutput("silent")`. `NULL` (the default) prints unless the level is silent. For a script that keeps many results and wants no output: see `joutput`.

Details

The dependent variable must be coded 0/1. If it is not, the function stops with a clear error message and suggests the appropriate `jrecode()` command.

Handles haven-labelled variables, registered dummy variables via `jdummy()`, and the `numeric`/`categorical` overrides in the same way as `jlm()`.

A predictor dropped because it is an exact linear combination of the others is named in a warning, as in `jlm()`. A model with no predictors, `y ~ 1`, is fitted too: its one coefficient is the log odds of the modeled category, the pseudo R-squared values are 0, and the Omnibus Test, which compares the model with that one, reads "None: the model has no predictors to test." (its p is `NA` in the returned object).

Transformed predictor terms in `formula` are computed automatically. A term that applies a function to a variable – `log(x)`, `sqrt(x)`, `exp(x)`, `I(x^2)`, `scale(x)`, an arithmetic expression, or a logical condition such as `I(x > 10)` – is evaluated once on the analysis data and enters the model as a single derived column named for the expression, so the coefficient table and the diagnostics report the term as written. This follows the base R formula convention; the terms supported inline are those that evaluate to one numeric or logical column. Terms that produce several columns (`poly(x, 2)`, spline bases) or a categorical result (`cut(x, 3)`) are not supported inline: create the derived variable as a column of the data first, then name that column in the formula. (The dependent variable must be a plain 0/1 dichotomy, so a transform applies to predictors, not the response.) A categorical variable may be compared but not transformed. `I(x == 2)`, `I(x %in% c(1, 3))` and, for a text variable, `I(x == "web")` each enter the model as a TRUE/FALSE predictor, as in `lm()`; an order comparison such as `I(x >= 3)` does too when `x` holds numbers. Arithmetic on a categorical `x` – `log(x)`, `I(x^2)` – is refused, because it would be computed from the category codes; register `x` with `jnumeric` if its codes are meant as numbers. A value or vector from your workspace may be named inside a computed term, as in `lm()`: `I(x > cutoff)` with `cutoff <- 10`. A data frame may not: write `y ~ x` with `data = MyData`, not `MyData$y ~ MyData$x`. A variable may not: jstats analyzes the variables of a data frame, so a vector holding one value for each case – `w` in `I(x * w)`, or an object reached with `$` such as `lst$w` – is added to the data frame first (`MyData$w <- w`), where `lm()` would read it from the workspace; the stop gives that line. A vector of any other length used value by value is refused, where `lm()` would repeat the shorter one; a set used with `%in%` may have any length.

In a formula, `^` applied to a single variable is not arithmetic: it is R's operator for interactions up to that order, so `x^2` enters the model as `x` alone. Write `I(x^2)` for the square. `jlogistic()` fits the model as written and then warns when a formula does this. A power that is not a number, `x^k`, stops with an error, as in `lm()`; write `I(x^k)`.

Value

Invisibly returns a list of class `jst_logistic` containing:

model The fitted `glm` object.

model_type Character string `logistic`.

model_frame The model frame used to fit the model.

formula_used The formula after dummy expansion.

coefficients Formatted coefficient table (data frame).

coefficients_raw Flat data frame of raw, full-precision coefficient statistics (one row per coefficient): `term` (machine key, shared with jlm), `b`, `SE`, `Wald`, `df`, `p`, `exp_b`, `exp_ci_lower` / `exp_ci_upper` odds-ratio CI bounds (present regardless of the `ci` display toggle), `beta_gelman` and `beta_product` (the full standardized coefficients, whichever `std` shows). Carries `beta_standardization` and `outcome` attributes.

fit_raw List of raw, full-precision model-level fit statistics: `ll_model`, `ll_null`, `deviance`, `null_deviance`, the omnibus likelihood-ratio test (`chi_sq`, `omnibus_df`, `omnibus_p`), Cox & Snell and Nagelkerke pseudo R-squared (`cox_snell_r2`, `nagelkerke_r2`), `aic`, and `n`.

nagelkerke_r2 Nagelkerke pseudo R-squared.

cox_snell_r2 Cox & Snell pseudo R-squared.

neg2ll -2 Log Likelihood.

aic Akaike Information Criterion.

omnibus Named vector: chi_square, df, p.

n Number of observations.

predicts Character string describing what the model predicts.

dummy_coef_names Names of dummy variable columns.

ref_cats Reference category descriptions.

dummy_groups Named list, one element per dummy-coded variable: `labels`, its categories as the coefficient table labels them, and `dummy`, the dummy column of each (`NA` for the reference). Used by `jplot()`.

vif Named numeric vector of VIF values, or NULL.

gvif The generalized VIFs, when the model has a set of dummy variables (a predictor of three or more categories): a data frame of `term`, `GVIF`, `Df` and `GVIF_adj` (`GVIF^(1/(2*Df))`), one row per predictor, as the VIF table prints them; NULL otherwise.

sample_info Pipeline and missing data counts.

Interactions

An interaction is entered in the formula with `*`, as in R: `y ~ x * z` fits `x`, `z` and their product, and the product's row in the coefficient table is labeled `x * z`. With a categorical predictor registered with `jdummy`, the interaction rows are grouped under a header naming the reference category, with one row per dummy variable, as the predictor's own rows are. Each row's b is the change in the log-odds slope of one variable per unit of the other (or, for a dummy row, the difference in slope between that category and the reference), and its Exp(b) the factor by which the other variable's odds ratio is multiplied per unit; the main-effect rows give the slopes where the other variable is zero. `jplot` on the result draws the probability plots.

Standardized coefficients

A logistic regression's coefficients are in each predictor's own units, so they cannot be compared across predictors measured differently. The Gelman beta column puts them on one footing (Gelman, 2008): the model is refit with every predictor centered, a continuous predictor also divided by two standard deviations and a binary predictor (a 0/1 variable, or a category's dummy variable) left on its one-unit scale. A continuous predictor's Gelman beta is then the change in the log-odds for a move from one standard deviation below its mean to one above, and a binary predictor's the difference between its two groups, so the two are comparable; `exp()` of a Gelman beta is the odds ratio for that move. The outcome is not standardized – it has no standard deviation to standardize by, which is why fully standardized logistic coefficients are contested and this column uses Gelman's scaling. It is the column `jlm(std = "gelman")` gives a linear model, computed the same way.

A computed term that is a power or a product of variables, such as `I(x^2)` or `I(x * z)`, is recomputed from the rescaled variables (Gelman, 2008, section 3.1), as an interaction written `x * z` is. In a model with an interaction or a squared term the predictors are centered, so a main effect's Gelman beta is its effect at the average of the variable it interacts with, while its b is the effect where that variable is 0; the two can differ in sign, and a note under the table says so. A variable that equals the product of two other predictors, or the square of one, is named in a note of its own, since the column treats it as an ordinary predictor.

`std = "product"` instead scales each column of the model as it stands – b times two standard deviations of the column, a binary column's b unchanged – the product of an interaction and a squared term included, the convention of software that computes a product as a new variable. On a model without such terms the two columns agree.

References

Gelman, A. (2008). Scaling regression inputs by dividing by two standard deviations. *Statistics in Medicine*, 27(15), 2865-2873.

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
# With explicit data frame -- Volunteer is already coded 0/1
jlogistic(Volunteer ~ Income + Age, data = community)

# The data frame may also be given positionally
jlogistic(Volunteer ~ Income + Age, community)

# An interaction, entered in the formula with *
jlogistic(Volunteer ~ Income * Age, data = community)

# A 1/2-coded dichotomy (Yes = 1, No = 2) must be recoded to 0/1 first
df <- community
df$OwnsHome01 <- jrecode(df, OwnsHome,
                         map = "1=1; 2=0", labels = "0=No; 1=Yes")
jlogistic(OwnsHome01 ~ Income + Age, data = df)

# Using juse() default
juse(community)
jlogistic(Volunteer ~ Income + Age)

# CATEGORICAL PREDICTORS
#
# Per-call: categorical = ... applies for one call only and does not
# persist.
jlogistic(Volunteer ~ Region + Age, categorical = "Region")

# The recommended approach for repeated analyses: register the variable
# with jdummy() before running jlogistic(). This sets categorical
# treatment persistently across subsequent analyses.
```

```
jdummy(community, Region)
jlogistic(Volunteer ~ Region + Age)

# To choose a non-default reference category:
jdummy(community, Region, ref = "West")
jlogistic(Volunteer ~ Region + Age)

# FORCING NUMERIC TREATMENT
#
# Use numeric = ... when a labelled variable should enter as a score.
jlogistic(Volunteer ~ Age + Education, numeric = "Education")

# jdummy(community, NULL) clears its registration -- not normally needed.
# You'd clear a default or registration only to undo a mistake, or -- as
# in this example -- to reset state for testing.
jdummy(community, NULL)
juse(NULL)
```

`jnumeric` *Register variables as numeric for analysis*

Description

`jnumeric()` tells jstats to treat one or more variables as numeric (continuous) wherever their analysis class matters, overriding the package's automatic structural guess. It is the counterpart to `jdummy` (categorical) and `jcount` (count): a variable carries exactly one registered intent at a time, so registering it as numeric clears any prior dummy or count registration. Registration changes no data and assigns nothing – you do not write `df <- jnumeric(...)`. It is stored for the session, keyed by the data frame's name; save the data frame in R format (.rds) to keep it across sessions.

Usage

```
jnumeric(data, ..., remove = FALSE, clear.all = FALSE)
```

Arguments

`data`	A data frame, or omitted to use the `juse` default. `jnumeric(NULL)` clears the numeric registrations on the `juse` default frame (or, with no default set, the only frame that carries them; if several do, it asks rather than wiping all). `jnumeric(data, NULL)` clears that one frame's numeric registrations. Called with no arguments, `jnumeric()` lists the session's numeric and count registrations. Registrations are stored under the data frame's name, so it is given as a name (or as a place such as `lst$d`); an expression that builds one is refused, with the two lines that name it first. The `NULL` and `remove = TRUE` forms work on a name whose data frame has since been removed.

`...`	One or more unquoted variable names to register. The registration applies to numeric variables and to haven-labelled variables that hold numbers. Any other type is refused, and none of the call's variables is then registered: convert a factor or a text variable to numbers first with `jencode`.
`remove`	Logical; if `TRUE`, remove the numeric registration for the named variables instead of adding it.
`clear.all`	Logical; if `TRUE`, clear numeric registrations on every data frame that carries them.

Details

The typical use is a small-range whole number that the structural classifier would treat as categorical (e.g. a 0-6 attitude item) but that you want analyzed as a continuous score.

Value

`invisible(NULL)`. Called for its side effect on the session registration notebook.

See Also

`jdummy`, `jcount`

Examples

```
# Treat a labelled Likert item as a continuous score (slope-per-unit)
jnumeric(community, Environment2)             # one labelled 1-5 item
jnumeric(community, Environment2, Environment4)  # several at once
jnumeric(data = community, Environment2)      # the data frame by name
juse(community); jnumeric(Environment2); juse(NULL)   # or the juse() default
jnumeric(community, Environment2, remove = TRUE) # unregister one
jnumeric()                          # list all registrations
jnumeric(community, NULL)           # clear community's numeric registrations
jnumeric(clear.all = TRUE)          # clear every frame's numeric registrations
```

`joptions` *Set or display session-level package options*

Description

Controls session-wide settings that affect how the package handles missing-value information and related conventions. `joptions` complements `joutput`: joutput governs output verbosity and tiering, while joptions holds session-wide conventions plus a small number of per-function display defaults (the `jcorr()` cell layout via `corr.layout`, and the `jfreq()` missing-value range detail via `missing.detail`), and the width at which runtime message prose wraps (`message.width`). Settings are read fresh on each function call: changing a setting after data has been loaded does not retroactively transform data already in memory. `jconvert` is the explicit transform path for data already in the workspace.

Usage

```
joptions(
  missing.convention = NULL,
  missing.convention.codes = NULL,
  data.dir = NULL,
  corr.layout = NULL,
  missing.detail = NULL,
  message.width = NULL,
  quiet = FALSE,
  ...
)
```

Arguments

missing.convention	One of "none", "spss", "stata", or "sas" (any capitalization is accepted). See Slots. A slot name arriving here unnamed and positionally is read instead as a status query on that slot (see Call patterns).
missing.convention.codes	Numeric vector, length 1 to 3. See Slots.
data.dir	Character string (length 1), or NULL. See Slots.
corr.layout	One of "wide" or "stacked", or NULL. See Slots.
missing.detail	One of "totals", "per_code", or "all", or NULL. See Slots.
message.width	One of "auto", "narrow", "medium", "wide", a whole number between 40 and 120, or NULL. See Slots.
quiet	Logical; default FALSE. When TRUE, joptions() applies the change silently, suppressing the settings echo, its pointer, the convention nudge, and the note that a data folder was created (the folder is created all the same). It has no effect on a status query, which makes no change to silence: the bare joptions() panel and a joptions("slot") query both print regardless.
...	Not used. An input joptions() does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Value

Invisibly returns NULL. Called for the side effect of updating session options and printing the settings panel – in full for a bare status query or a reset, as the named slots alone for a joptions("slot") query, or as an echo of the slots a setting call touched.

Slots

missing.convention Character, length 1. One of "none", "spss", "stata", or "sas". Default: "none", meaning no stated preference: loaded data is preserved as-is, and a call that would create a fresh missing-value declaration stops with a guided error asking you to choose a convention – the package never infers one. A set value states your working convention: it

supplies the target for fresh missing-value declarations on columns with no existing convention, becomes the default target for jconvert when to is not given, and is the reference point for the environment-scan notice (see below). Data already loaded is never changed by setting this; jconvert is the explicit transform path.

missing.convention.codes Numeric vector, length 1 to 3, whole numbers, no duplicates. Sign unconstrained. Default: c(-99, -98, -97). The recommended missing-value code set used by jconvert when translating Stata-style or SAS-style missing values into SPSS-style numeric codes: the first tag letter (.a or .A) takes the first code, the second the second, and so on, so a column carrying more distinct tags than there are codes cannot be converted. Also the source of the value the missing keyword creates in a jrecode or jencode map under an SPSS-style convention (the first code).

data.dir Character string (length 1), or NULL. Default: NULL. When NULL, jsave writes bare-filename saves to the working directory and jload searches the working directory. When set, names a folder (relative to the working directory) used as both the save target for bare-filename saves and as the first directory searched on bare-filename loads. If the folder does not exist it is created when the setting is made, with a note saying so unless quiet = TRUE (nested paths are created in full); a folder that cannot be created stops the call and leaves every setting as it was. jsave creates the folder again if it has been removed since. To clear a previously-set folder back to this default, pass data.dir = NULL or data.dir = "" (an empty string; see Call patterns). Filenames containing a directory separator (a forward slash, or a backslash on Windows) bypass this setting and are taken literally. Write the folder with forward slashes: a folder typed with single backslashes, which R reads as special characters, stops the call with nothing set.

corr.layout Character, length 1. One of "wide" or "stacked". Default: "wide". The default cell layout for jcorr when three or more variables are correlated: "wide" puts r and p on one line with N beneath; "stacked" stacks r, p, and N on three lines for a narrower table that fits more variables. A per-call layout argument to jcorr() overrides this. It lives here rather than in joutput because it is specific to one function's output, not a tiered analysis-content toggle.

missing.detail Character, length 1. One of "totals", "per_code", or "all". Default: "per_code". Governs how much of a declared missing-value RANGE jfreq spells out in its Missing block. "totals" collapses the whole band into one row; "per_code" prints one row per observed in-band value, at most 10, with the remainder gathered into a single line at the foot of the block; "all" prints every observed in-band value with no cap. Declared discrete codes always print in full at every setting – the cap applies only to values reached by a range. A per-call missing.detail argument to jfreq() overrides this. Like corr.layout it lives here rather than in joutput because it is specific to one function's output.

message.width Character or numeric, length 1. One of "auto", "narrow" (50 columns), "medium" (76), "wide" (90), or a whole number between 40 and 120. Default: "auto". The target width at which the package wraps runtime MESSAGE prose – errors, warnings and notes. "auto" follows the console pane, which R keeps current as the pane is resized, so it is resolved afresh for each message rather than fixed when it is set. A width outside 40 to 120 is refused rather than quietly adjusted. Analysis TABLES are not affected and do not reflow: prose can be re-wrapped without losing information, whereas breaking a correlation matrix mid-row would destroy the alignment that makes it readable.

Call patterns

`joptions()` Print the full settings panel: all six slots, whatever the convention. The `missing.convention.codes` row is labeled as SPSS-style; `jconvert` to SPSS-style missing values uses those codes under any setting.

`joptions("slot")` Print one slot and nothing else – `joptions("data.dir")`, or several at once with `joptions(c("data.dir", "corr.layout"))` – closed by a pointer to `joptions()` for the full panel. A slot name given WITHOUT an argument name is read as a query rather than a setting; the six slot names and the four `missing.convention` values share no string, so the two readings cannot collide. A named argument is always a setting, which leaves `joptions(missing.convention = "data.dir")` an error. Only the slots named are shown: a query pulls in no related slot, and `missing.convention.codes` appears whenever it is asked for, whatever the convention. Like the full panel, a query prints regardless of `quiet`.

`joptions(NULL)` Reset all slots to defaults, then print the full panel – everything changed.

`joptions(slot = value, ...)` Set one or more slots, then echo only what the call touched: the slots named, plus `missing.convention` whenever `missing.convention.codes` is set (the codes are read in light of the convention), and plus `missing.convention.codes` when `missing.convention` is set to `"spss"` (under any other convention the codes are dormant and the row is omitted), closed by a pointer to `joptions()` for the full panel. The other four slots are independent and are not pulled in. Passing `slot = NULL` as a named argument leaves that slot at its current value – useful for setting one slot without touching another – and echoes that unchanged value back. To reset a single slot to its default, pass the default value explicitly (e.g. `joptions(missing.convention = "none")`). `data.dir` is the one slot whose default is itself `NULL`, so for it a named `NULL` is that default: `joptions(data.dir = NULL)` clears the folder back to the working directory, as `data.dir = ""` does.

Environment-scan notice

Setting `missing.convention` to `"spss"`, `"stata"`, or `"sas"` triggers a one-time scan of `globalenv()` for data frames whose missing-value convention differs from the newly-set value. When mismatches exist, a notice lists the affected data frames and suggests `jconvert`: frames whose declared columns all carry one convention read "use X-style missing values", while frames with a genuine internal majority read "predominantly use". The notice is informational; nothing is changed. Plain data frames with no declared missing values – including the course datasets in their standard form – do not trigger the notice.

See Also

`joutput` for output-verbosity settings; `jstats` for the package overview.

Examples

```
joptions()                                    # show current settings
joptions("data.dir")                          # show one slot
joptions(c("data.dir", "corr.layout"))        # show two

# Setting a convention echoes the convention and its codes, then scans
# the workspace and notes any data frames whose missing-value
# convention differs (see the Environment-scan notice section):
```

```
joptions(missing.convention = "spss")             # set, echo, scan notice
joptions(missing.convention = "sas")              # SAS-style: .A, .B, ...
joptions(missing.convention.codes = c(-99, -98))      # set, echo, no scan
joptions(data.dir = file.path(tempdir(), "Data")) # set save/load folder
                                                  # (created if absent)
joptions(data.dir = NULL)                         # back to the working directory
joptions(message.width = 60)                      # wrap message prose at 60
joptions(message.width = "narrow")                # preset width (50 columns)
joptions(message.width = "auto")                  # follow the console pane
joptions(missing.convention = "stata",
         missing.convention.codes = c(-99, -98, -97)) # set both
joptions(missing.convention = "spss",
         missing.convention.codes = NULL)             # set mc, leave codes
joptions(NULL)                                    # reset all to defaults
```

---

joutput *Set session-level output verbosity*

Description

Controls what analysis functions display by default. Three preset levels are available, and individual toggles can override specific settings within any level; a fourth, `silent`, prints nothing from the analysis functions and is for a script that keeps their results. Per-call arguments on analysis functions always take precedence over joutput() settings.

Usage

```
joutput(
  level,
  effect.size = NULL,
  regression.ci = NULL,
  means.ci = NULL,
  posthoc = NULL,
  diagnostics = NULL,
  case.processing = NULL,
  case.processing.detail = NULL,
  case.processing.filter = NULL,
  variable.id = NULL,
  value.id = NULL,
  ref.categories = NULL,
  model.layout = NULL,
  missing.notice = NULL,
  digits = NULL,
  quiet = FALSE,
  ...
)
```

Arguments

`level` Character. One of `minimal`, `standard` (default), `full`, or `silent`. If omitted, prints the current settings. If `NULL`, resets to defaults (standard with no toggle overrides).

minimal Stripped-down output for power users. Core results only – no Case Processing table (a one-line N statement in its place), no variable labels, no reference categories, no effect sizes, no CIs.

standard Default. Suitable for teaching and routine use. Includes the Case Processing table when a filter or listwise deletion excluded cases (otherwise a one-line N statement), reference categories, effect sizes, and confidence intervals for means and mean differences (`jt`, `jaov`); regression coefficient CIs (`jlm`, `jlogistic`) are reserved for full. Variable labels are off by default (`variable.id = "names"`); request a label legend or in-table labels per call or via the `variable.id` toggle.

full Everything in standard plus a variable label legend (`variable.id = "legend"`), regression coefficient confidence intervals, post-hoc tests, and the Case Processing table on every call, with the per-code missing breakdown. Diagnostic output is not part of it: see `diagnostics`.

silent For a script that runs many analyses and keeps their results – a loop over models, a table built from several calls. The analysis functions (`jdesc`, `jfreq`, `jt`, `jaov`, `jcrosstab`, `jlm`, `jlogistic`, `jcorr`, `jalpha`, `jscreen`) print nothing, their notes included, and return their results as they always do; errors and warnings still show. Every other function prints as at standard, so a load's notice and the recode family's reminders stay, and the settings are standard's: a result made under silent holds what a standard call's would. A result is still there to keep: `m <- jlm(...)` saves it, as it does at any level. The same for one call: `print = FALSE` on any analysis function; and `print = TRUE` prints one call under a silent level.

`effect.size` Logical or NULL. Override the level's default for effect size display.

`regression.ci` Logical or NULL. Override the level's default for confidence intervals on regression coefficients (`jlm`, `jlogistic`). Off at minimal and standard, on at full.

`means.ci` Logical or NULL. Override the level's default for confidence intervals on means and mean differences (`jt`, `jaov`). Off at minimal, on at standard and full.

`posthoc` Logical or NULL. Override the level's default for post-hoc test display (jaov only).

`diagnostics` Logical, character vector, or NULL. The diagnostic output of the analysis functions – assumption checks and the like – which no level turns on. `TRUE` shows every diagnostic a function has; `FALSE` (the setting until one is made) shows none; a character vector shows the ones named, each function taking those that apply to it:

- `"levene"` – Levene's test for homogeneity of variance (`jt`, `jaov`);
- `"vif"` – the variance inflation factors (`jlm`, `jlogistic`);
- `"qq"` – the normal Q-Q plot of the residuals (`jt`, `jaov`, `jlm`);
- `"residuals"`, `"scale"` – the regression diagnostic plots (`jlm`);
- `"binned"` – the binned residual plot (`jlogistic`);

- "cooks", "leverage" – Cook's distance and residuals against leverage (jlm, jlogistic).

Name more than one with c(): joutput(diagnostics = c("levene", "vif")). A diagnostic prints with its brief interpretation (the note under a significant Levene's test, the lines under a VIF above 10) except at the minimal level, which prints the table alone. Unlike the other settings it is not cleared by a level call: joutput("full") leaves it as it was. joutput(NULL) turns it off with everything else. The same argument on jt(), jaov(), jlm() and jlogistic() sets it for one call.

case.processing

Three-state toggle for the Case Processing table – the block at the top of every analysis function's output (and of jscreen()'s) that accounts for the cases: the original N, each active filter (jcomplete, jsubset, per-call subset) with the cases it excluded, any cases dropped listwise by the analysis, and the N analyzed.

Every call states its N. What the toggle decides is the FORM: when the table does not print, a one-line N statement takes its place (Analysis N: 63 for the listwise functions; for jdesc, jfreq, and jcorr, the number of cases in the variable pool, adding the count complete on every variable when the per-variable Ns differ, and adding the excluded count whenever cases were excluded before the analysis). jscreen() prints no separate line: its header's Cases line states the count, adding the excluded count the same way (for example Cases: 53 (17 Excluded)).

- NULL (auto; the standard tier's default) prints the table only when it has an exclusion row to show: a filter is active (shown even when it excluded 0 cases, as the reminder that it is active) or listwise deletion dropped at least one case. Otherwise the N statement. A clean call with no filter therefore gets one line, not a table of zeros.
- TRUE (the full tier's default) prints the table on every call, even when its only rows are Original and the final N.
- FALSE (the minimal tier's default) never prints the table or its missing-data breakdown; the N statement only.

In every form, a listwise-deletion row appears only when it dropped at least one case; a clean analysis never shows a Auto-listwise 0 row. A grouped jdesc() call has the same kind of row for cases missing on its grouping variable, labeled by =, shown on the same terms. The missing-data breakdown beneath the table (or beneath the N statement) is governed separately by case.processing.detail.

case.processing.detail

Detail tier for the Case Processing Summary's missing-data breakdown: "none" (no bottom table), "totals" (one summed missing row per variable), or "per_code" (per declared missing-value code plus system-missing). The minimal tier defaults to "none", standard to "totals", full to "per_code". Each row counts the variable's missing cases in the original data (From N and its percent); when a filter step ran, the row also shows how many of those cases the filter removed (Filtered) and how many remain in the analysis pool (the second From N), where listwise deletion takes them. The breakdown lists the analysis variables and, after them, any variables in the jcomplete() list that the analysis itself does not use, each tagged (jcomplete() only): every case jcomplete() drops

is missing, and these rows say on which variable. The breakdown counts each variable's missing cases, so a case missing on two variables appears under both. A case that a `jsubset()` or per-call `subset` condition dropped because the condition could not be evaluated for it (a missing value in the condition's variables) is counted on that row of the upper table instead, as a `(k missing)` note after the condition, and gets no row here. Row form for the `jcomplete()`-only variables is set by `case.processing.filter`.

`case.processing.filter`
: Row form for the `jcomplete()`-only variables in the missing-data breakdown: `"list"` (one row per variable), `"collapse"` (a single `jcomplete()-only variables (k)` row counting the cases missing on at least one of them), or `"auto"` (named up to three variables, collapsed from four; the `jcomplete()` list of a long survey battery is the case that collapses). A lone variable is always named. The minimal tier defaults to `"collapse"`, standard to `"auto"`, full to `"list"`. Session-level only; the analysis functions take no per-call form of it.

`variable.id`
: Character or NULL. Variable label display mode, one of `"both"`, `"names"`, `"labels"`, `"legend"`, or `"legend.bottom"`. `"names"` shows variable names only; `"both"` shows `"name: label"`, with no labels block. `"labels"` replaces variable names with their labels in the analysis output itself (table rows, captions, crosstab dimnames, or `jplot` axis/legend titles) – best when labels are short. `"legend"` keeps names in place and prints a label legend at the function's mid position (for `jlm`/`jlogistic` between the coefficients and fit blocks; for `jfreq` under each variable's own table; elsewhere directly after the single table). `"legend.bottom"` keeps names in place and prints one consolidated legend at the very end of the output. The minimal and standard tiers default to `"names"`; the full tier defaults to `"legend"`. Not a logical – `TRUE`/`FALSE` are not accepted.

`value.id`
: Character or NULL. Value-label display mode for the categorical levels that appear in `jfreq` valid rows, the `jt`/`jaov` group descriptives, the `jcrosstab` axes, and the grouped `jdesc` headers. One of `"both"` (`"code: label"`, degrading to a bare code where a code has no label), `"values"` (the bare stored code), or `"labels"` (the value label, degrading to the bare code per code where none exists). `"legend"` and `"legend.bottom"` keep the bare code in the table and print a value-label legend after it (`"legend"` per-table, `"legend.bottom"` consolidated where multiple tables are produced). Variables with no value labels render identically under every mode, so this is a no-op for plain numeric data. The minimal tier defaults to `"labels"`; the standard and full tiers default to `"both"`. Distinct from `variable.id`, which governs the one-per-variable descriptive label. Not a logical.

`ref.categories`
: Logical or NULL. Override the level's default for the reference categories block (registered dummies).

`model.layout`
: Character or NULL. The order of `jlm()`'s and `jlogistic()`'s output: `"coefficients"` (every level's default) prints the Coefficients table first and the fit statistics after it; `"fit"` prints the fit statistics first. A level call puts it back to `"coefficients"`, as it puts back the other settings.

`missing.notice`
: Logical or NULL. Controls the notification about declared missing values that `jload()` emits for files whose variables carry them. `TRUE` prints it on every such

load; FALSE suppresses it; NULL (the default) leaves the level's setting in place. The standard and full levels print it; the minimal level suppresses it.

digits — Integer or NULL. Number of decimal places shown for continuous statistics in the analysis-function output tables (range 0-7; `digits = 0` prints whole numbers with no trailing decimal point). Every such statistic shows exactly this many places, trailing zeros included (0.100, not 0.1). Does not affect p-values, percentages, or integer quantities (counts, N, degrees of freedom), which keep their own fixed conventions, or the Min and Max columns of `jdesc()`, which show each variable's values to the decimal places its own data carry. All three preset levels default to 3.

quiet — Logical; default FALSE. When TRUE, joutput() applies the level/toggle change silently (nothing is printed). A bare joutput() status query, and a `joutput("setting")` query, always print regardless of quiet.

... — Not used. An input `joutput()` does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Details

Diagnostic output – Levene's test, the VIF table, the regression diagnostic plots – is set apart from the levels: it is off at every level and is turned on with `diagnostics`, here or in an analysis call.

Value

Invisibly returns NULL. Called for its side effect of setting session options and printing the settings panel – in full for a bare status query, a level change or a reset, as the named settings alone for a `joutput("setting")` query, or as an echo of the settings a setting call touched.

Call patterns

`joutput()` Print the full settings panel: the level and every setting, each marked `(override)` where its value differs from the level's default.

`joutput("setting")` Print one setting and nothing else – `joutput("digits")`, or several at once with `joutput(c("posthoc", "digits"))` – closed by a pointer to `joutput()` for the full panel. A setting name given WITHOUT an argument name is read as a query rather than as a level; the three level names and the setting names share no string, so the two readings cannot collide. Only the settings named are shown, and like the full panel a query prints regardless of `quiet`.

`joutput("level")` Set the level, clearing any earlier overrides, then print the full panel – a level moves most settings at once. Settings given in the same call are applied as overrides on the new level. `diagnostics` is the exception: it belongs to no level, so a level call leaves it as it was.

`joutput(setting = value, ...)` Set one or more settings within the current level, then echo only what the call touched, closed by a pointer to `joutput()` for the full panel. The three Case Processing settings (`case.processing`, `case.processing.detail`, `case.processing.filter`) are echoed together whenever one of them is set, because each is read in light of the others: `case.processing = FALSE` silences the other two, and `case.processing.detail = "none"`

silences `case.processing.filter`. Passing `setting = NULL` as a named argument leaves that setting as it is and echoes its current value.

`joutput(NULL)` Reset to the standard level with no overrides.

Session options

`joutput()` stores its settings through R's standard `options()` / `getOption()` mechanism, under two keys: `.jst_output_level` (the preset level) and `.jst_output_toggles` (a named list holding any per-slot overrides). `getOption(".jst_output_level")` returns the raw stored level – `NULL` until a level has been set, which the package reads as the `"standard"` default. `joutput(NULL)` clears both keys. The same values can be read or set with base R's `options()` directly; `joutput()` is the supported interface, adding validation and the settings display.

Color

jstats prints its titles in red and its status notes in yellow, and only where the color can be drawn: in the RStudio Console. Everywhere else the same text is printed plain – in RGui, in a terminal, in output captured with `sink()` or `capture.output()`, and in a knitr or Quarto document – because the codes that make the color would print there as stray characters.

One R option overrides the rule. `options(jstats.color = TRUE)` prints the colors wherever the output goes, which suits a console that draws color but is not RStudio's (Positron, VS Code, most terminals). `options(jstats.color = FALSE)` prints plain text everywhere, the RStudio Console included. Leaving it unset keeps the rule above. Like any R option it lasts for the session; put the line in your `.Rprofile` file to make it permanent.

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
joutput("standard")                       # effect sizes + means/diff CIs (jt, jaov)
joutput("standard", regression.ci = TRUE) # also show jlm/jlogistic coefficient CIs
joutput("full")                           # every statistic and every detail
joutput(diagnostics = TRUE)               # diagnostic output on, at any level
joutput(diagnostics = c("levene", "vif")) # only the diagnostics named
joutput(diagnostics = FALSE)              # and off again
joutput(digits = 2)                       # set one setting; echoes only that one
joutput("digits")                         # show one setting
joutput()                                 # show current settings
getOption(".jst_output_level")            # the raw option behind the level
joutput(NULL)                             # reset to defaults
```

jplot *Visualize jst_* result objects or plot variables directly from a data frame*

Description

Unified plotting function. Can be called in three ways:

Usage

```
jplot(x, which = "core", ...)

## Default S3 method:
jplot(
  x,
  ...,
  by = NULL,
  type = NULL,
  line = FALSE,
  equation = TRUE,
  r2 = TRUE,
  band = "ci",
  subset = NULL,
  labels = NULL,
  numeric = NULL,
  categorical = NULL,
  count = NULL,
  facet = NULL
)

## S3 method for class 'jst_lm'
jplot(
  x,
  which = "core",
  focal = NULL,
  at = "zero",
  equation = TRUE,
  r2 = TRUE,
  ...
)

## S3 method for class 'jst_logistic'
jplot(x, which = "core", focal = NULL, at = "zero", ...)

## S3 method for class 'jst_ttest'
jplot(x, which = "core", ...)
```

```
## S3 method for class 'jst_anova'
jplot(x, which = "core", ...)

## S3 method for class 'jst_corr'
jplot(x, which = "core", ...)

## S3 method for class 'jst_crosstab'
jplot(x, which = "core", ...)

## S3 method for class 'jst_desc'
jplot(x, which = "core", ...)

## S3 method for class 'jst_freq'
jplot(x, which = "core", ...)
```

Arguments

x	A result object from one of the package's analysis functions (result-object form), or a data frame (data-first form).
which	Character vector. `core` (default), `all`, or one or more specific plot names valid for the object's class. (Result-object form only.)
...	Additional arguments: for the result-object form these are passed to class-specific methods; for the data-first form these are unquoted variable names (1 or 2).
by	Unquoted variable name for group-coloring (data-first form).
type	Character. Plot type override for the data-first form. One of `histogram`, `density`, `bar`, `scatter`, `box`, `grouped_bar`. If NULL (default), auto-detected from variable types; `density` is never the automatic choice.
line	Controls a line overlay on data-first scatter plots. One of `FALSE` (default; no line), `TRUE` (alias for `lm`), `lm`, `loess`, `connect`.
equation	Logical. If TRUE (default), displays the equation in the subtitle for `line = "lm"` scatter plots (data-first form) or `jst_lm fit` plots (result-object form).
r2	Logical. If TRUE (default), displays R-squared in the subtitle alongside the equation.
band	Character. Uncertainty band type for `line = "lm"` scatter plots. One of `ci` (default; 95% confidence band for the mean, flares at the ends), `pi` (95% prediction interval for individual observations), `see` (constant-width band at +/- t*SEE; useful for teaching homoskedasticity), `none`.
subset	Optional unquoted logical expression to filter cases for this call only (data-first form). Written in R syntax and checked the way `jsubset()` checks a filter: `subset = NOT(Age < 40)`, for example, is refused with the corrected `subset = !(Age < 40)` shown (see `jsubset` for a translation table).
labels	Character or NULL. Variable label display mode (data-first and formula forms): one of `"both"`, `"names"`, `"labels"`, `"legend"`, or `"legend.bottom"`. `"names"` uses variable names as axis/legend titles; `"labels"` uses each variable's label as its axis/legend title instead (falling back to the name when unlabelled) and prints no console legend; `"legend"` and `"legend.bottom"` keep names on the

axes and print a console label legend. "both" is accepted but currently renders as "names" on plots (the "name: label" form for plot titles is deferred to a later phase). NULL (default) defers to joutput()'s variable.id setting. Not a logical.

numeric — Optional. Plotted variables to treat as continuous for this call (the per-call counterpart of jnumeric()), named bare or in quotes, several with c(): numeric = Education. In jplot() a variable's class chooses the geometry, so this forces numeric handling (histogram for a single variable; scatter / numeric axis in the formula and two-variable forms). Applies to the plotted variables only, not the by grouping variable.

categorical — Optional. Plotted variables to treat as categorical for this call (the per-call counterpart of jdummy() for plotting purposes), named bare or in quotes, several with c(). Forces categorical geometry (bar for a single variable; box / categorical axis in the formula form). A variable cannot be listed in both categorical and numeric/count.

count — Optional. Plotted variables to treat as counts for this call (the per-call counterpart of jcount()), named bare or in quotes, several with c(). A count is numeric-like for plotting, so it draws the same as numeric; it is provided for symmetry with the other analysis functions.

facet — Unquoted variable name for one panel per group (variable-list form: a histogram, density plot, bar chart or grouped bar chart), bare or in quotes. See Details.

focal — Unquoted name of the independent variable to place on the x-axis for jst_lm / jst_logistic fit and probability plots. Defaults to the first independent variable in the model that is not dummy-coded, since a fitted line needs a numeric axis. A dummy-coded categorical variable is named as the formula names it (focal = Region), and is drawn with its categories along the axis: the predicted outcome (or probability) at each with its 95% confidence interval, the cases behind them.

at — Character string or named list specifying where non-focal independent variables are held when drawing the fitted line in jst_lm / jst_logistic methods. One of zero (default), mean, mixed (categorical at 0, interval at mean), or a named list list(Var1 = value, ...). A term computed from the focal variable alone, or from it and variables held – a square, I(x^2); a log, log(x + 1) – is not held: it is computed from the focal variable at each point of the line, so the line of a model with a squared term is a curve.

Details

Result-object form: Pass a result object returned by one of the package's analysis functions. Produces appropriate plots for each class of result (see valid plot names below).

Formula form (for plots that distinguish DV from IV): Pass a formula as the first argument, followed optionally by a data frame, given either positionally or as data = : jplot(WellbeingScore ~ Income, data = community) is the same plot as jplot(WellbeingScore ~ Income, community). Used for scatterplots and boxplots, consistent with the formula syntax of jlm(), jaov(), and jt(). The DV on the left of ~ goes on the y-axis; the IV on the right goes on the x-axis. Only single-IV formulas are supported here; for multi-IV models, fit with jlm() and pass the result to jplot().

Each side names a variable as it stands: a computed term such as `log(Income)` or `I(Age > 40)` is refused, so create it as a variable of the data first.

Variable-list form (for distributions and counts): Pass a data frame followed by one or two unquoted variable names. The data frame is given either positionally or as `data =`: `jplot(data = community, Age)` is the same plot as `jplot(community, Age)`. Used for histograms (1 numeric), bar charts (1 categorical), and grouped bar charts (2 categorical). Calls that would otherwise autodetect to a scatter or boxplot produce a helpful error directing you to the formula form.

A labelled variable with a few categories, such as Region – six values or fewer with at least one of them labelled, or whole numbers from 0 to 6 – is drawn as categories when nothing says otherwise: alone, as a bar chart; beside another variable in the variable-list form, as one of a grouped bar chart's two variables or as a boxplot's groups (the error then gives the formula line); on the right of a formula, as the groups of a boxplot. This is for the plot only: the analysis functions read such a variable as numbers unless it is registered or asserted (see `jdummy`). `numeric =` or `count =`, a `jnumeric()` or `jcount()` registration, a `type`, or `line`, `band`, `equation` or `r2` (which only a scatter plot draws) keeps the numbers.

Supports pipeline integration (`jsubset`, `jcomplete`, per-call `subset`), grouping via `by =`, and regression lines with equation/R-squared/band annotations.

Valid plot names by class (for the result-object form):

- `jst_lm`: `fit`, `predicted`, `effects`, `coef`, `residuals`, `qq`, `scale`, `cooks`, `leverage`
- `jst_logistic`: `probability`, `logit` (the same fitted line on the log-odds scale, where it is straight: the linear predictor and its 95% band against the focal variable, the cases as a rug along the axis, those with the outcome at the top and the others at the bottom; at a dummy-coded focal variable's categories, the log-odds at each), `roc`, `calibration`, `binned`, `cooks`, `leverage`, `coef`
- `jst_ttest`, `jst_anova`: `box` (the default), `qq` (a normal Q-Q plot of the standardized residuals: each case's distance from its group's mean, or for a paired test each pair's difference from the mean difference)
- `jst_corr`: `heatmap`, `scatter` (scatter requires exactly 2 variables in the correlation)
- `jst_crosstab`: `bar`

The shortcut keyword `core` (default) produces a curated default set for the class; `all` produces every plot the class supports.

For a `jst_lm` result, `effects` draws one plot per variable of the model: a term computed from a variable (`I(x^2)`, `log(x + 1)`, `I(x * z)`) is drawn in that variable's plot, following it along the axis, with the other variables held as `at` sets them. A dummy-coded categorical variable is one variable, drawn as one plot: its categories along the axis, labeled as the coefficient table labels them, with the predicted outcome at each and its 95% confidence interval. A variable the fit dropped as an exact linear combination of the others has no effect to draw. For a model with no predictors (`y ~ 1`), the plots that need a predictor, or that are drawn against the fitted values or the leverage, are left out with a note.

Valid plot types for the data-first form: `histogram`, `density`, `bar`, `scatter`, `box`, `grouped_bar`. A density plot is the histogram's alternative for a numeric variable – its kernel density estimate, one curve, or one per group of `by =` laid over each other – and is drawn only when asked for with `type = "density"`.

In the variable-list form `facet =` draws one panel per group of the variable named, with the groups' value labels as the panel titles: the alternative to `by =` when the groups are many, since colors laid over each other stop being readable past a handful. The two can be combined, `by =` coloring the groups of one variable within each panel of another. A scatterplot or boxplot by groups takes `by =` in the formula form; `facet =` is refused there.

Valid `line` values: `FALSE` (default), `TRUE` (alias for `lm`), `lm`, `loess`, `connect`.

Valid `band` values: `ci` (default confidence band around the regression line, flares at the ends), `pi` (prediction interval for individual observations, wider), `see` (constant-width +/- t*SEE band illustrating the homoskedasticity assumption), `none` (no band).

Value

Invisibly, a single `ggplot` object if one plot is produced, or a named list of `ggplot` objects if multiple are produced (result-object form). Invisibly returns the `ggplot` object for the data-first form.

Methods (by class)

- `jplot(default)`: the default method: a scatter or box plot from a formula (`DV ~ IV`), or a histogram or bar chart from a data frame and one or more variables.
- `jplot(jst_lm)`: diagnostic, coefficient (forest), and fitted-effect plots for a `jlm()` linear-regression result.
- `jplot(jst_logistic)`: predicted-probability (S-curve) and coefficient plots for a `jlogistic()` result.
- `jplot(jst_ttest)`: a group-comparison box plot for a `jt()` result, with the group means marked; `which = "qq"` draws the normal Q-Q plot of its residuals.
- `jplot(jst_anova)`: a group-comparison box plot for a `jaov()` result, with the group means marked; `which = "qq"` draws the normal Q-Q plot of its residuals.
- `jplot(jst_corr)`: a heat-map of the correlation matrix for a `jcorr()` result, or a scatter plot for a single pair.
- `jplot(jst_crosstab)`: a grouped bar chart of cell counts for a `jcrosstab()` result.
- `jplot(jst_desc)`: (planned) direct plotting of a `jdesc()` result is not yet available; this method points you to the data-first form, for example `jplot(data, Variable)`.
- `jplot(jst_freq)`: (planned) direct plotting of a `jfreq()` result is not yet available; this method points you to the data-first form, for example `jplot(data, Variable)`.

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
# Result-object form
m <- jlm(WellbeingScore ~ Income + Age, community)
jplot(m)                            # core diagnostics + fit plot
jplot(m, which = "coef")            # coefficient forest plot
jplot(m, which = "fit", focal = Age, at = "mean")
```

```
# Formula form (scatter and box)
jplot(WellbeingScore ~ Income, community)               # scatter
jplot(WellbeingScore ~ Income, community, line = "lm")  # + regression line
jplot(WellbeingScore ~ Income, community, line = "lm", band = "see")
jplot(WellbeingScore ~ Income, community, by = Volunteer, line = "lm")

# Boxplot: a labelled variable with a few categories, such as Region,
# is drawn as groups on the right of the formula (for the plot only)
jplot(WellbeingScore ~ Region, community)
# categorical = asserts it in any variable; numeric = keeps the codes
jplot(WellbeingScore ~ Region, community, categorical = "Region")

# Variable-list form (distributions and counts)
jplot(community, Age)                      # histogram
jplot(data = community, Age)               # the same, data frame as data =
jplot(community, Region)                   # bar chart
jplot(community, Region, Volunteer)        # grouped bar chart

# Using juse() default (omit the data frame in either form)
juse(community)
jplot(WellbeingScore ~ Income)               # scatter
jplot(WellbeingScore ~ Income, line = "lm")  # + regression line
jplot(Age)                                   # histogram
jplot(Region, Volunteer)                     # grouped bar chart
```

`jrecode` *Recode a variable with explicit value mapping and optional labels*

Description

`jrecode()` recodes a variable using a simple map string that specifies how old values should be converted to new values. It is designed for situations where you need to collapse categories, change numeric codes, or recode dichotomies. Variable and value labels are handled automatically.

Map and labels rules can also produce missing values: plain system NA via the `NA` / `System` / `SYSMIS` aliases, or tagged missing values (`.a` through `.z`, or `.A` through `.Z`) when the active convention is Stata or SAS. See *Missing values in the map* below for the canonical patterns under each convention.

Usage

```
jrecode(data, orig.var, map, labels = NULL, convention = NULL, ...)
```

Arguments

`data`	A data frame containing the original variable.
`orig.var`	The variable to recode (unquoted, e.g. `AgeGroup`; a quoted name is accepted).

map

A quoted string specifying the recode rules, using the format "old=new" with rules separated by semicolons. Multiple old values mapping to the same new value are separated by commas on the left side.

An optional else clause controls what happens to values not covered by the map:

- No else clause: the function stops with a message if any values are left unmapped, so you can fix the map before proceeding.
- else=NA (also else=System or else=SYSMIS): unmapped values are deliberately set to system NA.
- else=copy: unmapped values are carried across unchanged.
- else=.a (or any other marker, .a through .z; Stata or SAS convention only): unmapped values are set to that Stata-style or SAS-style missing value.

Individual values can also be mapped to system NA using the same aliases: "-5=NA", "-5=System", or "-5=SYSMIS".

The same aliases may also appear as an OLD value, converting plain NA cells to a code: "NA=-98; else=copy" recodes every NA to -98 (declare the code afterward with jdeclare_missing()). NA may be combined with numeric old values in one rule ("NA,-99=-98") and may be named in at most one rule. Only plain NA cells are affected; tagged missing values are never converted by an NA rule. Under Stata or SAS convention the target may itself be a marker: "NA=.a".

Under Stata or SAS convention, values can be mapped to Stata-style or SAS-style missing values by their markers: "-99=.a; -98=.b".

A lettered marker can also be an OLD value, recoding the cells that carry it (either case: .a also names a SAS-style .A): ".a=-99" gives them a number, ".a=NA" makes them plain NA, and ".e=.c" re-letters them, merging them into .c if the variable already has it (a marker on the right needs Stata or SAS convention, as always). Markers and numbers may share a rule ("9,.a=-99"). A marker the map does not name is kept, with its label, whatever the else setting.

The right-hand side may also be the word missing (case-insensitive): the value is converted to your working convention's own missing form – a tagged marker under the Stata or SAS convention, or the first code from the missing.convention.codes setting in joptions under the SPSS convention, declared on the result automatically. The same map string therefore works under every setting: "8=missing; else=copy". The NA rule composes with it ("NA=missing" converts plain NA cells the same way), and labels = "missing=Refused" labels whatever the keyword produced. If the column already carries the other convention's markers while your missing.convention setting is set, the call stops and shows both resolutions rather than guessing.

Under the SPSS convention the code the keyword uses must be free: if the column already declares it, discretely or inside a declared range, the existing declaration is reused rather than added to; if it is an ordinary value in the column that the map leaves in place, or a value the map itself assigns, the call stops rather than sweep those cases into missingness; and if the declared codes the result keeps already fill what SPSS allows (three codes, or a range plus one code), the call stops and offers a declared code instead. Under the Stata or SAS convention the keyword always uses the first marker (.a, or .A), so a map that

also assigns that marker itself stops rather than merge two missing categories onto one marker; the message offers the next free marker, or the same marker for both when that is what you mean. When the variable already carries that marker – data from Stata often has its own `.a` – the recoded cases join it, and a note says so and gives the line that keeps them apart (the next free marker). Under every convention, a `labels = "missing=..."` that differs from the label the marker or code already has stops, since it would rename the cases already there: the message gives the line that keeps the two apart and the line that merges them on purpose, keeping the existing label.

Examples:

- `"1=1; 2=0"`
- `"1=1; 2,3=2; 4,5=3; else=NA"`
- `"1=1; 2=0; else=copy"`
- `"-5=System; else=copy"`
- `"NA=-98; else=copy"`
- `"3=1; 4=2; else=.a"` (Stata or SAS convention only)
- `".a=-99; .b=-98; else=copy"` (a Stata-style variable to numeric codes, declared on the result)
- `"8=missing; else=copy"` (any convention)

`labels` Optional. A quoted string specifying value labels for the new variable, using the format `"code=Label Text"` with rules separated by semicolons. If supplied, these labels are used for the codes they name; the labels of declared missing values the result keeps (see Details) ride alongside them unless the string labels the same code. A label may be written in quotes, with either mark, right after its equals sign, and a semicolon inside the quotes is then part of the label: `"1=Yes; 9=\"Refused; not asked\""`. An unquoted label reads as it is written.

The left side of each rule may be a numeric code or, under Stata or SAS convention, a missing-value marker (`.a` through `.z`, or `.A` through `.Z`). A marker's label is stored on the marker itself, not on a numeric code. It may also be the word `missing`, labelling whatever the map's `missing` target produced: `labels = "missing=Refused"`.

If omitted, the function attempts to transfer value labels automatically from the original variable. This works when the original variable has value labels and the mapping is one-to-one (no categories are collapsed). When categories are collapsed, labels cannot be transferred automatically and a note is printed.

Example: `"1=Male; 0=Female"` or `".a=Refused; .b=Don't know"`.

`convention` Optional. One of `"spss"`, `"stata"`, `"sas"`, or `NULL` (default); any capitalization is accepted. Controls whether missing-value markers (`.a` through `.z` or `.A` through `.Z`) are accepted in the map and labels arguments. Marker letters are matched case-insensitively; the stored markers take the convention's letter case (lowercase Stata-style under `"stata"`, uppercase SAS-style under `"sas"`). Also decides what the `missing` keyword produces. Inert when neither a marker nor `missing` appears in either argument.

`...` Not used. An input `jrecode()` does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

When NULL, the convention is resolved from the missing.convention setting in joptions; if that is also unset, the call stops with a guided error asking you to choose – the package never infers a convention. Most users set the convention once at the top of a session via joptions() (or in their .Rprofile) rather than supplying this argument on every call. See ?joptions for details.

Details

The function accepts haven-labelled, plain numeric, and logical variables. Factor, character, and date/time variables are refused with a message naming the fix — recoding works with a variable's numeric values, so convert these to numeric first.

The variable label from the original variable is carried across automatically with "(recoded)" appended. If the original variable has no variable label, the variable name is used instead.

Value labels are handled in three ways, in order of priority:

1. If labels is supplied, those labels are used as-is.
2. If labels is omitted and the original variable has value labels, they are automatically transferred to the new codes — provided the mapping is one-to-one (no collapsing). For example, recoding 1/2 to 1/0 will carry "Yes" and "No" across to the new codes automatically.
3. If categories are collapsed (multiple old values map to one new value), automatic transfer is not possible and a note is printed directing you to supply labels manually. Recoding onto a value that else=copy also carries across ("5=4; else=copy") collapses two categories the same way.

Labels move with lettered markers as they do with numbers: a marker recoded to a number or to another marker takes its label along, and a value recoded to a marker gives the marker its label. Where a label would land on a value that keeps a label of its own ("-99=-98", or ".b=.a" on a variable with both markers), the value keeps its own; several different labels landing on one new value are all dropped. Labels that move onto a declared missing value or a marker also ride beside a supplied labels string, as the labels of kept declared values do.

NA values in the original variable are carried across as NA unless the map names NA as an old value (for example "NA=-98"); the else setting never converts NA. An NA rule affects plain NA cells only — tagged missing values (Stata-style or SAS-style) are declared missings and are preserved with their tags unless the map names them as old values (".a=-99"). Declared SPSS-style missing values on the original variable are likewise preserved, whether declared as discrete codes or as a range: a declared code the map does not name, and every value inside a declared range, are carried onto the result with the declaration and their labels under every else setting, and a note names the values the data hold (map them to NA explicitly to convert them instead). A declared code the map names is recoded like any other value; a range always carries whole, so a value the map recodes into the range is missing on the result.

A declared missing value the map moves onto a new code stays missing when the new code looks like a coded missing value (the check jload describes under *Coded missing values*): "-99=-88; else=copy" on a variable declaring -99 declares -88 on the result, with the label of -99 and a note saying so, and -99 is no longer declared. A lettered marker recoded to a number works the same way (".a=-99"). A new code that does not pass the check becomes an ordinary value – deliberately so for "-99=6", which makes refusals a category, but also for a negative code on a variable with large values, such as incomes, where the check finds nothing unusual; declare such a code with jdeclare_missing(). So does a new code that also receives ordinary values. A result that would

hold both such a declared code and a lettered marker, or more declared codes than SPSS allows, stops the call with a message.

A variable holds one convention. When the original variable's SPSS-style declaration survives the map (a range always does; a discrete code does unless the map names it), Stata-style or SAS-style markers in the map or labels are refused with a pointer to `jconvert()`. A map that names every declared code (`"-99=.a; -98=.b; else=copy"` under Stata convention) leaves nothing SPSS-style behind and is the one-column migration pattern.

Values that merely look like coded missing values (e.g. -99, -9, 999) but are not declared are never changed on their own. Left unmapped with no `else` clause, they are named in the error that stops the call, with a nudge toward `jdeclare_missing()`; carried across by `else=copy`, they draw an advisory note at the full `joutput` tier. Recoding values *to* a code that looks like a missing value draws a note suggesting the declaration.

If the map does not include an `else` clause and there are unmapped values in the variable, the function stops with a message listing the unmapped values so you can fix the map before proceeding.

If the map specifies values that do not exist in the original variable, nothing is recoded for them and an advisory note says so at the full `joutput` tier. This helps catch typos in the map string.

Missing values in the map. The package supports three conventions for representing declared missing values, and the syntax for producing them from `jrecode()` depends on which one is active. A convention becomes active via `joptions(missing.convention = ...)` or this call's `convention` argument; with neither set, a call that produces declared missing values stops with a guided error asking you to choose.

Under **SPSS convention**, declared missing values are real numeric codes carrying metadata that flags them as missing. The two-step canonical pattern is:

```
df$EducR <- jrecode(df, Education,
                    map    = "1,2=1; 3=2; 4,5=3; -99,-98=-99",
                    labels = "1=High school or less; 2=Some college; 3=Degree")
jdeclare_missing(df, EducR, codes = c(Refused = -99), modify = TRUE)
```

The `jrecode()` call assigns the numeric code `-99`; the subsequent `jdeclare_missing()` call attaches the label and flags `-99` as missing. Labeling `-99` inside the `labels` argument is unnecessary — `jdeclare_missing()` owns that label.

The same two-step pattern serves data whose missingness arrived as plain `NA` (data born in R, or read from a CSV): `map = "NA=-98; else=copy"` converts the NA cells to the numeric code, and `jdeclare_missing()` declares it.

Missingness that arrived as Stata-style or SAS-style markers takes one step: naming the markers moves them to codes that are declared on the result (`map = ".a=-99; .b=-98; else=copy"`), as described above. `jconvert(to = "spss")` does the same for every such variable in a data frame, taking the codes from your setting.

Under **Stata convention**, declared missing values are typed missing cells marked with Stata-style tags (`.a` through `.z`). The single-call canonical pattern names the same codes:

```
df$EducR <- jrecode(df, Education,
                    map    = "1,2=1; 3=2; 4,5=3; -99,-98=.a",
              labels = "1=High school or less; 2=Some college; 3=Degree; .a=Refused")
```

Under Stata convention, `jdeclare_missing()` is not needed for this pattern — `jrecode()` handles both the value recoding and the Stata-style missing-value labeling in one call. Naming the declared codes is what makes the call legal on a column that carries SPSS-style missing values: a map that left `-99` and `-98` in place (`else=.a` alone) would put a marker beside them and is refused.

SAS convention works the same way with SAS-style missing values (`.A` through `.Z`). Markers in the map and labels are matched case-insensitively, and the stored markers take the active convention's letter case: lowercase under Stata convention, uppercase under SAS convention.

Writing these markers while the active convention is SPSS raises an error naming the mismatch and the two ways out: restate the markers as numeric codes to stay in SPSS convention, or switch convention with `joptions(missing.convention = ...)` (or with this call's `convention` argument). The error does not rewrite the call for you: the SPSS-style codes would have to be taken from the `missing.convention.codes` setting in `joptions`, which cannot be known to be free of collision with values already in the column. The two-call SPSS-style pattern is documented above.

Value

A `haven_labelled` vector with the recoded values, variable label, and (if supplied or auto-transferred) value labels applied. The function returns the values rather than changing `data`: assign the result to a column to keep it. Two assignment forms do that job:

- Preserving (recommended): `MyData$AgeGroupR <- jrecode(MyData, AgeGroup, map = "...")` adds a new column and keeps the original, so the two can be compared with `jfreq()` before the original is retired.
- Overwrite: `MyData$AgeGroup <- jrecode(MyData, AgeGroup, map = "...")` replaces the variable in place. Nothing is wrong with this once the recode is verified, but the original values are gone.

SPSS users will recognize the pair: `RECODE ... INTO new.` is the preserving form and bare `RECODE` the overwrite.

See Also

`jdeclare_missing` for declaring missing values on a column after a recode (the SPSS-style canonical pattern).

`jrelabel` for applying labels to an existing variable after a recode.

`joptions` for the session-level `missing.convention` setting.

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
# Recode with explicit labels (a 1/2 dichotomy to 0/1)
df <- community
df$OwnsHome01 <- jrecode(df, OwnsHome,
                         map    = "1=1; 2=0",
                         labels = "0=No; 1=Yes")

# The data frame may also be given by name
df$OwnsHome01 <- jrecode(data = df, OwnsHome, map = "1=1; 2=0",
```

```
                        labels = "0=No; 1=Yes")

# Collapse categories (must supply labels)
df$RegionR <- jrecode(df, Region,
                      map    = "1,2=1; 3,4=2",
                      labels = "1=North or South; 2=East or West")

# Use else=copy to carry unspecified values across unchanged
df$EducR <- jrecode(df, Education,
                    map    = "5=4; else=copy",
                    labels = "4=Bachelor's degree or higher")

# Use else=NA to deliberately drop unspecified values to system NA
df$EducR2 <- jrecode(df, Education,
                     map    = "4=1; 5=1; else=NA",
                     labels = "1=College degree")

# Convert a specific coded missing value to system NA
df$EducR3 <- jrecode(df, Education, map = "-99=System; else=copy")

# Give plain NA cells a codable value, then declare it. Declaring on a
# plain column needs a chosen missing-value convention (the package
# never infers one), so choose it first:
joptions(missing.convention = "spss")
df$AgeR <- jrecode(df, Age, map = "NA=-98; else=copy")
df <- jdeclare_missing(df, AgeR, codes = c("Not recorded" = -98))

# Stata convention: Stata-style missing values in map and labels, named
# by their markers (single call; convention = "stata" scopes the choice
# to this call only).
# Education carries SPSS-style missing values (-99, -98); naming them in
# the map moves them to the marker in the same call.
df$EducR4 <- jrecode(df, Education,
                     map    = "1,2=1; 3,4,5=2; -99,-98=.a",
                     labels = "1=No college; 2=College; .a=Refused",
                     convention = "stata")

# SAS convention: the same single-call pattern; markers are matched
# case-insensitively and store in uppercase (.A)
df$EducR5 <- jrecode(df, Education,
                     map    = "1,2=1; 3,4,5=2; -99,-98=.a",
                     labels = "1=No college; 2=College; .a=Refused",
                     convention = "sas")

# Using juse() default
juse(df)
df$RegionR2 <- jrecode(Region, map = "1,2=1; 3,4=2",
                       labels = "1=North or South; 2=East or West")
```

---

jrelabel *Apply variable and value labels to a variable*

Description

jrelabel() attaches a variable label and/or value labels to any variable in a data frame. It is designed as a simple label applicator — it does not recode values, convert types, or compare variables. Use it to add labels after a recode, to fix missing labels, or to label any variable that needs them.

It has two forms. The one-variable form, jrelabel(MyData, Q1, labels = ...), returns the labeled variable for you to assign to a column, or with modify = TRUE labels the variable in the data frame directly. The list form, jrelabel(MyData, c(Q1, Q2, Q3), labels = ..., modify = TRUE), gives several variables the same value labels in one call – the way a set of survey items sharing one response scale is labeled – and always changes the data frame itself, so it takes modify = TRUE.

A variable label (var.label) can be attached to a variable of any type — the variable is returned unchanged apart from the new label, so dates stay dates, factors stay factors, and text stays text. Value labels (labels) can be applied to haven-labelled, plain numeric, and logical variables; logical values are stored as 1 (TRUE) and 0 (FALSE). Factor, character, and date/time variables cannot carry value labels, and jrelabel() refuses the labels argument for these; for factor and text variables the message names jencode as the fix.

jrelabel() never rebuilds the variable it is given. Existing value labels, SPSS-style missing values (na_values / na_range), Stata-style missing values, and the variable's class all pass through untouched unless an argument you supply replaces them: new value labels replace the existing ones (as VALUE LABELS does in SPSS), except that a declared missing value keeps its label unless the new set names it, as jrecode()'s labels argument keeps them, and a note names the labels kept; a new variable label replaces the old one. To change a declared value's label, name it in the new set, e.g. labels = "1=Low; 2=High; -99=No answer". On a column with Stata-style or SAS-style missing values, a marker that no case carries is declared only by its label, and it stays declared; name it by its marker to relabel it, e.g. labels = "1=Low; .c=Refused".

Both the labels and var.label arguments are optional in the one-variable form. If neither is supplied, the function returns the variable unchanged.

Usage

```
jrelabel(
  data,
  var,
  labels = NULL,
  var.label = NULL,
  vars = NULL,
  modify = FALSE,
  ...
)
```

Arguments

data	A data frame containing the variable.
var	The variable to label (unquoted, e.g. StatusR; a quoted name is accepted), or a c() of two or more for the list form (see the section below).
labels	Optional. A quoted string specifying value labels using the format "code=Label Text" with rules separated by semicolons. Accepted on haven-labelled, plain

numeric, and logical variables only. A label may be written in quotes, with either mark, right after its equals sign, and a semicolon inside the quotes is then part of the label: `"1=Yes; 9=\"Refused; not asked\""`. An unquoted label reads as it is written.

Examples:

- `"1=Yes; 0=No"`
- `"1=Employed; 2=Unemployed; 3=Student; 4=Retired"`

`var.label` Optional. A quoted string to use as the variable label (the description shown by `jdesc()`, `jfreq()`, etc.). If omitted, any existing variable label is preserved. If the variable has no existing label, no variable label is set. One-variable form only.

`vars` Optional. For the list form: the variables' names in quotes, a stored or computed list such as `vars = items` or `vars = c("Q1", "Q2")`. Names typed without quotes go after the data frame, in a `c()`.

`modify` Logical. `TRUE` labels the variable(s) in the data frame directly, so no assignment is needed; the data frame must be given by its name (or set with `juse()`). Optional in the one-variable form, which still returns the variable; required in the list form. Default `FALSE`.

`...` Not used. An input `jrelabel()` does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Value

The one-variable form: the variable with the requested labels applied, returned invisibly (nothing prints at the console), with or without `modify = TRUE`. The variable keeps its class: haven-labelled input stays haven-labelled with any declared SPSS-style or Stata-style missing values intact; plain numeric and logical input becomes `haven_labelled` when value labels are applied; any other type is returned unchanged apart from the labels. Without `modify = TRUE`, assign the result to a column to keep it (at the standard and full output levels a note says so after each call). Two assignment forms do that job:

- Preserving: `MyData$VarNameL <- jrelabel(MyData, VarName, ...)` adds a labeled copy and keeps the original as it was.
- Overwrite (the usual choice here): `MyData$VarName <- jrelabel(MyData, VarName, ...)` labels the variable in place. Labels change how values are shown, not the values, so there is normally nothing to keep a copy of.

SPSS users will recognize the pair from `RECODE ... INTO new.` and bare `RECODE`; for labels, SPSS's `VALUE LABELS` and `VARIABLE LABELS` always label in place, the overwrite form – which `modify = TRUE` gives directly.

The list form: the changed data frame, invisibly.

The list form

A `c()` of two or more variables after the data frame – or a stored list of names in quotes, `vars = items` – applies the same value labels to each variable, in the data frame itself:

```
jrelabel(MyData, c(Q1, Q2, Q3),
         labels = "1=Never; 2=Sometimes; 3=Often", modify = TRUE)
```

The SPSS equivalent is `VALUE LABELS Q1 Q2 Q3 1 'Never' 2 'Sometimes' 3 'Often'.` The form takes `modify = TRUE` and stops without it, offering the call with it added: a call that changes several variables returns the whole data frame, and assigning that to one column (`MyData$Q1 <- jrelabel(...)`) would put the data frame inside the column. It also returns the changed data frame, so `e <- jrelabel(d, c(Q1, Q2), ..., modify = TRUE)` labels `d` and keeps a copy in `e`. Every variable is checked before any is changed, so a call that stops changes nothing. A variable label names one variable and is refused in this form; give each its own call. A list in `vars =` follows these rules whatever its length, so a script that builds its list of names behaves the same when the list holds one name; `c(Q1)` after the data frame is the one-variable form.

See Also

jrecode for recoding values with optional labels in a single step.

jstats for the package overview, workflow conventions, and complete function listing.

Examples

```
# Add value labels after a recode
df <- data.frame(Status = c(1, 2, 1, 2, 1, 2))
df$StatusR <- ifelse(df$Status == 1, 1, 0)
df$StatusR <- jrelabel(df, StatusR, labels = "1=Yes; 0=No",
                       var.label = "Status (recoded)")

# The same, labeling the variable in the data frame directly
jrelabel(df, StatusR, labels = "1=Yes; 0=No", modify = TRUE)

# The data frame may also be given by name
df$StatusR <- jrelabel(data = df, StatusR, labels = "1=Yes; 0=No")

# Add just a variable label
df$StatusR <- jrelabel(df, StatusR, var.label = "Employment Status")

# Add just value labels
df$StatusR <- jrelabel(df, StatusR, labels = "1=Yes; 0=No")

# Label a date variable (the variable stays a Date)
df$Enrolled <- as.Date(c("2024-01-15", "2024-02-01", "2024-01-20",
                         "2024-03-05", "2024-02-14", "2024-01-30"))
df$Enrolled <- jrelabel(df, Enrolled, var.label = "Enrollment date")

# Several items that share one response scale: the list form
sv <- data.frame(Q1 = c(1, 3, 2), Q2 = c(2, 2, 3), Q3 = c(3, 1, 1))
jrelabel(sv, c(Q1, Q2, Q3), labels = "1=Never; 2=Sometimes; 3=Often",
         modify = TRUE)

# The same from a stored list of names
items <- c("Q1", "Q2", "Q3")
jrelabel(sv, vars = items, labels = "1=Never; 2=Sometimes; 3=Often",
```

```
          modify = TRUE)

# Using juse() default
juse(df)
df$StatusR <- jrelabel(StatusR, labels = "1=Active; 0=Inactive")
juse(NULL)
```

`jsave` *Save a data frame to a file*

Description

`jsave()` writes a data frame to a file. Supports SPSS (`.sav`), Stata (`.dta`), SAS interchange (`.xpt`), Excel (`.xlsx`), CSV (`.csv`), and R's native `.rds` format.

The file format is determined entirely by the file extension you provide — for example, `"mydata.sav"` saves as SPSS, `"mydata.dta"` saves as Stata, and `"mydata.xlsx"` saves as Excel. Changing the extension changes the format.

By default, `jsave()` writes bare-filename saves to the working directory, matching base R's `saveRDS()` and `write.csv()`. To save into a subfolder, set `joptions(data.dir = "...")` once per session (or in `.Rprofile`). Filenames containing a directory separator (`/`, or a backslash) bypass this setting and are taken literally.

If the `data` argument is omitted, the default data frame set by `juse()` is used.

Usage

```
jsave(
  data,
  file,
  overwrite = FALSE,
  preserve.declarations = TRUE,
  clean.names = FALSE,
  ...
)
```

Arguments

`data`	A data frame (unquoted). If omitted, uses the default set by `juse()`.
`file`	Character string. The filename with extension (e.g. `"mydata.sav"`) or a full file path. Use forward slashes in file paths. If the extension is omitted, `.rds` is used and added to the filename.
`overwrite`	Logical. If `TRUE`, overwrites an existing file without prompting. If `FALSE` (default), prompts for confirmation in interactive sessions. In non-interactive sessions, stops with an error. In a script, state it explicitly: `jsave(mydata, "mydata.rds", overwrite = TRUE)`. Otherwise, if the file already exists and the script was run by pasting or with RStudio's Run button, the call stops with an error.

`preserve.declarations`
: Logical. If `TRUE` (the default), missing-value declarations are written as they stand; formats that cannot store them (notably Excel and CSV) drop the metadata, and SPSS-style codes such as -99 then read back as ordinary numbers, while a missing-value form that .sav, .dta, or .xpt cannot represent stops the save before anything is written. If `FALSE`, every declared missing value – an SPSS-style code, or a Stata-style or SAS-style marker – is made plain NA before writing, in the file only (the data frame is not changed), and a note gives the number of cells. This happens before the .sav, .dta, and .xpt checks, so a form those formats cannot represent no longer stops the save. Mirrors the `preserve.declarations` argument of `jload`. The haven package and its documentation call these user-defined missing values.

`clean.names`
: Logical. For `.sav`, `.dta` and `.xpt`. If `FALSE` (the default), a variable name the format cannot hold stops the save, listing every such name, why, and the name it would be given. If `TRUE`, those names are changed in the file only, the way `jload` changes names R cannot use; the data frame keeps its own names, each old name becomes the variable's label in the file where it has none, and a note lists the changes.

`...`
: Not used. An input `jsave()` does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Details

File paths: Use forward slashes (`/`) in file paths. If you copy a path from Windows File Explorer, replace the backslashes with forward slashes. R does not accept single backslashes in file paths: inside quoted text it reads a backslash and the letter after it as one special character. `jsave()` stops on such a path, before anything is written, and shows it written with forward slashes.

File location:

- If the path contains a directory separator, the file is saved to that exact location; the folder must already exist.
- If the path is a bare filename and the `data.dir` setting in `joptions` is set, the file is saved to that folder (`joptions()` creates it when the setting is made; it is created again here if it has been removed since).
- If the path is a bare filename and the `data.dir` setting in `joptions` is unset (the default), the file is saved to the working directory.

Format notes:

- SPSS (`.sav`) and Stata (`.dta`) preserve variable labels and value labels.
- Excel (`.xlsx`) and CSV (`.csv`) do not preserve variable or value labels.
- R format (`.rds`) preserves the data frame exactly as it exists in R, including all attributes.
- R format (`.rds`) also carries the active classification registrations (see `jdummy`, `jnumeric`, `jcount`, and `jlikert`), which `jload` restores. Other formats cannot store them, and a note says so when any are lost.
- Before writing `.sav`, `.dta`, or `.xpt`, `jsave()` checks for anything the format cannot store and stops, naming every problem at once: complex, raw, list, or POSIXlt columns; missing values

of a form the format cannot hold (Stata-style in `.sav`, SPSS-style in `.dta`, either in `.xpt`); and, for `.sav`, more than three declared codes, or a range plus more than one code, on one column, and a text variable's declared missing value longer than 8 bytes, which the file would cut short so its cells read back as data. No file is written until the problems are fixed. A missing-value form the format cannot hold can also be saved as plain NA with `preserve.declarations = FALSE`, which the message offers beside the conversion.

- SPSS (`.sav`), Stata (`.dta`), SAS interchange (`.xpt`), CSV (`.csv`) and Excel (`.xlsx`) have no missing value for text: a text variable's missing cells are saved as blank text (in Excel, empty cells), and read back as blank, not missing. A note says so when it happens. R format (`.rds`) keeps them missing. In a CSV or Excel file the same holds for a factor, which is written as its labels.
- Variable names each format can hold: SPSS (`.sav`), a letter or @ first, then letters, digits and . _ $ # @, not ending in a period, at most 64 bytes, not a word SPSS reserves (ALL, AND, BY, EQ, GE, GT, LE, LT, NE, NOT, OR, TO, WITH), and no two that differ only in case; Stata (`.dta`), a letter or underscore first, then letters, digits and underscores, at most 32 characters, not a word Stata reserves (such as `long`, `in`, `byte`); SAS (`.xpt`), the letters A to Z, digits and underscores, not a digit first, at most 32 characters, and no two that differ only in case. A name outside these rules stops the save, naming every one at once, unless `clean.names = TRUE`.
- Labels each format shortens: SPSS keeps 256 bytes of a variable label and 120 of a value label; Stata keeps 80 characters of a variable label and of the data frame's label; SAS interchange keeps 40 characters of the data frame's label. A note says what was shortened in the file; the data frame keeps its labels whole.
- An Excel sheet holds 1,048,575 cases below the names row and 16,384 variables; a larger data frame stops with lines that save it in another format.
- Inside an `.xpt` file the SAS dataset is named after the file, changed to a SAS name when the file's name is not one (a note says so).
- A CSV file is written in UTF-8. When it holds a character beyond plain ASCII (an accented letter, a curly apostrophe, a dash), it starts with the UTF-8 byte-order mark, as Excel's own "CSV UTF-8" does, so that Excel opens it with those characters intact: without the mark, Excel on Windows reads a CSV file in Windows-1252. A file of plain ASCII has no mark. `jload()` reads the mark as the start of a UTF-8 file.
- Stata files are written as version 14 format.
- Legacy Excel format (`.xls`) is not supported for saving. Use `.xlsx` instead.

Value

Invisibly returns `NULL`. Called for its side effect of writing a file to disk.

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
# A runnable save into R's session temporary folder
jsave(community, file.path(tempdir(), "community.sav"), overwrite = TRUE)
jsave(data = community, file.path(tempdir(), "community.sav"),
```

```
      overwrite = TRUE)                  # the data frame given by name

## Not run:
# The file extension determines the format ---
# the same data frame can be saved in any supported format
jsave(community, "community.sav")         # SPSS
jsave(community, "community.xlsx")        # Excel
jsave(community, "community.csv")         # CSV
jsave(community, "community.rds")         # R format

# Using juse() default
juse(community)
jsave("community.sav")

# Full file path
jsave(community, "C:/Output/community.sav")

# Stata and SAS formats cannot carry community's SPSS-style missing-value
# declarations -- convert first (jsave() pre-flights this and says so)
jconvert(community, to = "stata", modify = TRUE)
jsave(community, "community.dta")                  # Stata
jconvert(community, to = "baseR", modify = TRUE)
jsave(community, "community.xpt")                  # SAS interchange

## End(Not run)
```

`jscreen` *Data screening overview*

Description

Provides a quick overview of a data frame for screening. A red "Data Screening" title is printed first (followed by a Case Processing table when a filter is active), then a short header block (case and variable counts, cases with missing data, variables with outliers), followed by up to three tables: a Variable Types table (Base R storage type, the jstats analysis-role class, an optional sub-class, an optional classification source, distinct-value counts, and optional central- tendency columns), a Missing Data & Outliers table, and – when variable labels are shown – a Variable Labels table last. Handles haven-labelled and date/time variables gracefully.

Usage

```
jscreen(
  data,
  ...,
  outlier.sd = 3,
  subset = NULL,
  variable.id = NULL,
  value.id = NULL,
```

```
  types = TRUE,
  issues = TRUE,
  r.type = FALSE,
  stats = FALSE,
  digits = NULL,
  print = NULL
)
```

Arguments

data	A data frame, or a single variable: a column of a data frame or a vector.
...	Optional unquoted variable names to screen. If omitted, all variables in the data frame are screened. None when data is a single variable. A c() of names, bare or quoted, is read as the names.
outlier.sd	Numeric. Number of standard deviations from the mean to flag as potential outliers (Numeric-class variables only). Default is 3.
subset	An optional unquoted logical expression (e.g. Group == 1) to subset cases for this call only. Applied after jcomplete and jsubset. Does not affect other function calls. The cases it excludes are listed in the Case Processing table. Written in R syntax and checked the way jsubset() checks a filter: subset = NOT(Age < 40), for example, is refused with the corrected subset = !(Age < 40) shown (see jsubset for a translation table).
variable.id	Character or NULL. Variable label display mode: one of "both", "names", "labels", "legend", or "legend.bottom". "names" shows variable names only; "both" shows "name: label"; "labels" shows labels in each table's Variable column (best for short labels); "legend" and "legend.bottom" keep names and print a label legend after the tables. NULL (default) defers to joutput()'s variable.id setting. Not a logical.
value.id	Not supported by jscreen(). The function does not display value labels, so passing this argument is an error. It exists only to return a clear message rather than report value.id as a variable that was not found. Leave at NULL (default).
types	Logical. If TRUE (default), prints the Variable Types table. Set FALSE to suppress it.
issues	Logical. If TRUE (default), prints the Missing Data & Outliers table, which lists only the variables that actually have missing data or flagged outliers (clean variables are omitted). Set FALSE to suppress the table entirely. Suppressing types, issues, and labels together leaves only the header block.
r.type	Logical. If TRUE, adds a "Base R Type" column (numeric / haven_labelled / factor / character / date-time) to the Variable Types table, showing each variable's storage type alongside its jstats class. Default is FALSE: the storage type is expert detail (its main signal is "this variable carries value labels / came from SPSS, Stata, or SAS"), so it is opt-in rather than shown by default. The returned data frame always includes it regardless of this setting.
stats	Logical or character. Adds central-tendency columns to the Variable Types table for numeric-like variables. FALSE (default) shows none; TRUE shows both Mean and Median; "mean" or "median" shows only that one. Numeric and

Count variables show both; a numeric dichotomy shows its raw mean and a blank median; N-category and other non-numeric variables are blank. The returned data frame always includes Mean and Median regardless of this setting.

`digits` Integer or NULL. Number of decimal places for the Mean and Median columns. NULL (default) defers to `joutput()`'s `digits` setting (default 3).

`print` Logical or NULL. `FALSE` prints nothing – no table, no title, no note – and returns the result as every call does; errors and warnings still show. `TRUE` prints even under `joutput("silent")`. `NULL` (the default) prints unless the level is silent. For a script that keeps many results and wants no output: see `joutput`.

Details

The jstats Class column reports how the package treats each variable in analyses (Numeric, Categorical, Numbers-as-text, Date-time, Unsupported), in contrast to the Base R Type column's storage view; the same classification gates outlier screening, so only Numeric-class variables are SD-screened and the Outliers cell is left blank for the rest. Zero counts are shown blank so only affected variables carry numbers; a column (or the whole Missing/Outliers table) is omitted entirely when nothing is flagged, and the header count lines explain the omission.

A variable R cannot treat as a column of values – a list column (a tibble's, or the geometry column of a spatial data frame), a raw one, or a column that is itself a data frame – is listed as Unsupported and screened no further; it does not stop the screening of the rest.

Text cells with no text – empty, or holding only spaces or tabs – are counted apart from missing values, never among them. When the screened variables hold any, the header gains a "Cases with blank text" line and the Missing Data & Outliers table a Blank and a % Blank column. Whether a blank is absent data or a real answer depends on the file; see `jfreq`.

A text variable whose every value is a date written in one form – as 2015-04-01, 1-Apr-15, 28NOV2007 or 01/02/2015 – has the sub-class "date (unparsed)": a CSV file stores dates as text, and as text each date is a category. A footnote under the Variable Types table gives the `as.Date()` line that makes it a date (two lines when its values could be day/month or month/day). When the dates have two-digit years, a note says so: `as.Date()` puts 69 to 99 in the 1900s and 00 to 68 in the 2000s without a warning, so a birth date written 03-Jul-68 becomes 2068. The note gives the `range()` line that shows the years. Month names are read as R reads them in an English locale.

When at least one variable's class comes from a registration (jnumeric, jcount, or jdummy) rather than the structural guess, a Source column appears. It reads as an exception-marker: "registered" is shown against the registered variables and the structurally classified rows are left blank, so the registrations stand out at a glance. (The returned data frame still records the literal tier for every row.) Set `stats = TRUE` (or "mean" / "median") to add central-tendency columns for the numeric-like variables: Numeric and Count variables show Mean and Median, while a numeric dichotomy shows the raw mean of its stored codes and a blank median. A numeric dichotomy coded other than 0/1 (e.g. the 1/2 Group-4 coding) is flagged with a "*" on its sub-class cell, since its raw mean is not a proportion; the marker shows even when `stats` is off, surfacing the recode need. A dichotomy registered with `jnumeric()` or `jcount()` is Numeric, has no sub-class, and carries no marker.

When variable names are supplied, only those variables are screened. When omitted, all variables in the data frame are screened. Settings stored with `jsubset()` and `jcomplete()`, and a `subset` expression, apply to the whole data frame before the named variables are taken, so they may refer to variables that are not screened.

Given a single variable instead of a data frame, as in `jscreen(community$Income)`, jscreen() screens that variable. A column of a data frame is screened in its frame, exactly as `jscreen(community, Income)` screens it: the frame's `jsubset()` and `jcomplete()` settings and its registrations apply. Any other vector is screened on its own. A second variable, and a `subset` condition naming another variable, need the data frame, and each stops with that form of the call.

With a `juse()` default set, a name the default data frame has is that frame's variable, as in every jstats function, even when a separate object of the same name exists in the workspace, and a line under "Using default data frame" says so. A separate object is read only when the default frame has no variable of that name, or when no default is set.

Those filters are accounted for as in the analysis functions: a Case Processing table between the title and the header lists the original count, the cases each filter excluded, and the count remaining, which the header's Cases line repeats. With no filter active no table prints, and the Cases line states the count. The `case.processing` setting of `joutput()` applies: at the minimal tier, or with `case.processing = FALSE`, the table is left out and the Cases line carries the excluded count instead (for example `Cases: 53 (17 Excluded)`); at the full tier, or with `case.processing = TRUE`, the table prints even when no filter is active.

Value

Invisibly returns a data frame of the screening results, with one row per variable and columns including the Base R type, the jstats `Class` and `SubClass`, the classification `Source` ("registered" or "structural"), distinct-value count, missing count and percentage, the count and percentage of blank text cells (`Blank`, `Pct_Blank`), the outlier count (NA for non-Numeric variables), and the `Mean` and `Median` (NA where not meaningful: Median is NA for dichotomies, and both are NA for non-numeric-like variables). The returned values are the raw counts; only the printed tables blank zeros and omit clean rows.

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
# With explicit data frame
jscreen(community)
jscreen(community, outlier.sd = 2.5)

# The data frame may also be given by name
jscreen(data = community)

# Show the Base R storage type column
jscreen(community, r.type = TRUE)

# Add Mean and Median columns for numeric-like variables
jscreen(community, stats = TRUE)

# Suppress tables (header block only)
jscreen(community, types = FALSE, issues = FALSE)

# A single variable
```

```
jscreen(community$Income)

# Using juse() default
juse(community)
jscreen()
jscreen(Income, Age, WellbeingScore)
jscreen(Income, Age, WellbeingScore, subset = Volunteer == 1)
```

`jsubset` *Set, activate, deactivate, or clear a per-dataset case-selection expression*

Description

`jsubset()` sets a persistent case-selection expression that is applied automatically by jstats analysis functions when the default data frame (set by `juse()`) is in use. This is analogous to the SPSS FILTER command.

The expression is stored per dataset, so switching `juse()` between datasets preserves each dataset's setting independently.

The expression applies whenever the matching dataset is used, regardless of whether it was supplied via `juse()` or specified explicitly in a function call. To bypass it temporarily without losing it, use `jsubset(off)` before the analysis and `jsubset(on)` afterward (or `jsubset(d, off)` and `jsubset(d, on)` for a named dataset). This matches the SPSS FILTER / USE ALL convention.

Expressions use standard R logical operators: `==`, `!=`, `<`, `<=`, `>`, `>=`, `&` (and), `|` (or), `!` (not), `xor()`, and `%in%`. Two habits carried over from commercial statistical software are caught with an error that shows the corrected form: a single `=` where `==` was meant, and the words `AND` / `OR` / `NOT` used as operators. Not every spelling can be caught, because R's own parser reads the call before jstats does: `jsubset(Gender = 1)`, `(Gender = 1) & (Age < 40)` and `NOT(Age < 40)` are all shown their corrected form, but a keyword between two conditions, `Age < 40 AND Gender == 1`, is refused by R itself with a parser message ("unexpected symbol") that jstats cannot replace. The same checks run on the `subset =` input of the analysis functions. See the translation table below.

The expression must give one TRUE or FALSE for every row of the dataset. `jsubset()` runs it once when set and refuses anything else – a single value (`TRUE`, or an aggregate such as `mean(Score) > 5`), numbers, text, or the wrong number of values – with an error that shows a corrected form. A filter that cannot be run at all is refused in the same way, and nothing is stored: one that names a variable or an object that is not found (a misspelled variable is the usual cause), and one that stops with an error of R's own, which is reported. Everything the filter names must therefore exist when `jsubset()` is called. The same checks run when the filter is applied, so a filter that was valid when set but can no longer be applied – a variable or an object it names has been removed, or it has stopped matching the dataset – stops the analysis rather than running on the wrong rows; that error names both ways out, `jsubset(d, off)` and `jsubset(d, NULL)`.

A filter normally names only columns of the data frame, and such a filter can never fall out of step with the data. A filter may also refer to a value in your workspace, such as a cutoff (`Age < cutoff`), a set of codes (`Region %in% keep_regions`) or a summary (`Age > mean(ages)`). It may not refer to a vector of values for the cases: jstats analyzes the variables of a data frame, so a keep/drop indicator

computed separately is added to the data frame as a column first (`clinic$Keep <- clinic$Stress > 3`, then `jsubset(clinic, Keep == TRUE)`), where it stays matched to its rows and the data frame's other settings reach it. A filter naming such a vector is refused, with that line. Any other vector compared with a variable value by value is refused too, where R would repeat the shorter one.

A text variable's blank cells, which `jfreq()` tabulates in a row labeled `<blank>`, are selected by what they hold, not by that label: `jsubset(d, trimws(Source) == "")` keeps the cases whose Source is empty or holds only spaces, and `Source != ""` with `trimws()` drops them. `Source == "<blank>"` matches nothing.

Usage

```
jsubset(data, expr, clear.all = FALSE, ...)
```

Arguments

`data`	Optional data frame. If supplied, the expression is stored on that dataset specifically. If omitted, the dataset set by `juse()` is used.
`expr`	A logical expression (e.g. `Age < 40 & Gender == 1`), or one of the following special values: `off` Deactivate the setting but remember the expression. `on` Reactivate a previously deactivated setting. The filter is run once first, as it was when set; one that can no longer be applied is refused, and stays off. `NULL` Clear the setting entirely (forget the expression). Each acts on the `juse()` default dataset when `data` is omitted (`jsubset(off)`, `jsubset(NULL)`), or on the named dataset when it is given (`jsubset(d, off)`, `jsubset(d, NULL)`). With no default set, `jsubset(NULL)` clears the one dataset that carries a setting, and asks you to name one when several do. To clear every dataset's setting at once, use `clear.all = TRUE`. If `expr` and `data` are both omitted, prints the current jsubset status, which says so when a stored filter can no longer be applied. A setting is stored under its dataset's name, so the named `off` and `NULL` still work after that dataset has been removed, or its name given to something that is not a data frame; `on` is refused there, since the filter cannot be checked without its dataset. `jsubset()` takes one condition. Two conditions separated by a comma (`jsubset(Age < 40, Gender == 1)`) are refused, with the call that joins them: `jsubset(Age < 40 & Gender == 1)`. `off`, `on` and `NULL` are given alone, after the dataset when one is named. Given along with a condition (`jsubset(d, Age < 40, on)`), they are refused, with the call that sets the filter and the call that acts on the stored one.
`clear.all`	Logical. If `TRUE`, clears the jsubset setting on every dataset; use on its own, `jsubset(clear.all = TRUE)`. This is the same grammar as the registration functions (`jdummy()`, `jnumeric()`, `jcount()`, `jlikert()`): `NULL` clears one dataset, `clear.all = TRUE` clears them all.
`...`	Not used for variables. Catches a condition typed with a single `=`, `jsubset(Gender = 1)`, so that the corrected `jsubset(Gender == 1)` can be shown in place of R's own "unused argument" error.

Value

Invisibly returns `NULL`. Called for its side effect.

Writing a filter in R

Stata already writes conditions the way R does. SPSS and SAS do not, and their forms translate as follows:

SPSS or SAS	R
`Gender = 1`	`Gender == 1`
`Gender NE 1`	`Gender != 1`
`Age GE 40`	`Age >= 40`
`Age < 40 AND Gender = 1`	`Age < 40 & Gender == 1`
`Age < 40 OR Age > 60`	`Age < 40 | Age > 60`
`NOT (Age < 40)`	`!(Age < 40)`
`MISSING(Age)`	`is.na(Age)`
`ANY(Region, 1, 3, 5)`	`Region %in% c(1, 3, 5)`

See Also

jstats for the package overview, workflow conventions, and complete function listing.

Examples

```
juse(community)
jsubset(Age < 40)                        # Set using juse default
jsubset(community, Age < 40)             # Explicit dataset
jsubset(data = community, Age < 40)      # ... given by name
jsubset(Age < 40 & WellbeingScore > 50)  # Compound condition
jsubset(off)                             # Deactivate
jsubset(on)                              # Reactivate
jsubset(community, off)                  # Deactivate on a named dataset
jsubset(community, on)                   # ... and reactivate it
jsubset()                                # Check status
jsubset(NULL)                            # Clear default dataset's setting
jsubset(community, NULL)                 # Clear a named dataset's setting
jsubset(clear.all = TRUE)                # Clear every dataset's setting

# juse(NULL) clears the default dataset -- not normally needed. You'd
# clear a default or registration only to undo a mistake, or -- as in
# this example -- to reset state for testing.
juse(NULL)
```

jsum *Compute a row-wise sum across multiple variables*

Description

`jsum()` computes the sum of values across multiple variables for each case (row) in the data frame. This is typically used to create composite scores from a set of related items (e.g. summing 6 survey items into a total scale score).

By default, cases with any missing values receive `NA`. Use the `min.valid` argument to allow partial sums — for example, `min.valid = 1` returns the sum of available values as long as at least one item is non-missing.

Variables can be listed individually or using colon notation to select a range of consecutive columns (e.g. `Attitude1:Attitude6`).

Usage

```
jsum(data, ..., min.valid = NULL, var.label = NULL)
```

Arguments

`data`	A data frame, or omit to use the `juse()` default.
`...`	Unquoted variable names. Use colon notation (e.g. `Attitude1:Attitude6`) to select a range of consecutive columns.
`min.valid`	Integer (optional). The minimum number of non-missing values required to compute a sum. If a case has fewer non-missing values, the result is `NA`. If omitted, all values must be non-missing (equivalent to setting min.valid to the number of variables).
`var.label`	Character string (optional). A variable label to attach to the result. If omitted, an auto-generated label is used.

Value

A numeric vector the same length as `nrow(data)`, suitable for assigning to a new column: `MyData$Total <- jsum(Var1, Var2, Var3)`.

See Also

`javg` for computing row-wise means.

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
# Set the default data frame (so you can omit it in function calls)
juse(community)

# Sum three variables (all must be non-missing)
community$EnvTotal <- jsum(Environment1, Environment3, Environment4)

# Sum with partial data allowed (at least 2 non-missing)
community$EnvTotal <- jsum(Environment1, Environment3, Environment4,
                           min.valid = 2)

# Sum using colon range for consecutive columns
community$EnvTotal <- jsum(Environment1:Environment5)

# Mix colon ranges and explicit names (e.g. after reverse-coding an item)
community$Environment2R <- jrecode(community, Environment2,
                                   map = "1=5; 2=4; 3=3; 4=2; 5=1")
community$ScaleTotal <- jsum(Environment1, Environment2R,
                             Environment3:Environment5)

# With a custom variable label
community$ScaleTotal <- jsum(Environment1:Environment5,
                             var.label = "Environment Scale Total")

# With an explicit data frame (instead of using juse default),
# positional or named
community$EnvTotal <- jsum(community, Environment1, Environment3,
                           Environment4)
community$EnvTotal <- jsum(data = community, Environment1, Environment3,
                           Environment4)

# juse(NULL) clears the default dataset -- not normally needed. You'd
# clear a default or registration only to undo a mistake, or -- as in
# this example -- to reset state for testing.
juse(NULL)
```

---

jt *Independent samples or paired samples t-test*

Description

Runs a t-test and prints formatted group descriptives and test results. By default, runs the traditional Student's independent samples t-test assuming equal variances. Optional parameters provide Welch's correction, paired samples, effect size (Cohen's d), Levene's test, and confidence interval for the mean difference. Handles haven-labelled, numeric, and factor grouping variables. For haven-labelled variables, numeric codes are displayed alongside labels in the group descriptives table.

Usage

```
jt(
  formula,
  data,
  paired = FALSE,
  welch = FALSE,
  effect.size = NULL,
  diagnostics = NULL,
  ci = NULL,
  subset = NULL,
  variable.id = NULL,
  value.id = NULL,
  case.processing.detail = NULL,
  full = FALSE,
  digits = NULL,
  print = NULL,
  ...
)
```

Arguments

formula	A formula of the form `DV ~ Group`. A transformed term such as `log(DV)` is computed automatically: the test and the descriptive output both use the transformed values.
data	A data frame containing variables referenced in `formula`.
paired	Logical. If TRUE, runs a paired samples t-test. Cases are paired by position: the i-th case in one group is matched with the i-th case in the other, so the two groups must have equal sample sizes. A pair is dropped from the analysis when either member is missing (matching how commercial statistical software handles paired comparisons), and a note reports how many pairs were dropped. Default is FALSE.
welch	Logical. If FALSE (default), runs Student's t-test (equal variances assumed). If TRUE, runs Welch's t-test, which needs at least 2 cases in each group; Student's test can include a group of one case. Ignored when paired = TRUE.
effect.size	Logical or NULL. If TRUE, prints Cohen's d. If NULL (default), defers to `joutput()` session setting.
diagnostics	Logical, character vector, or NULL. If TRUE, prints Levene's test for homogeneity of variance, with a note under it when the test is significant (see "Unequal variances"), and draws a normal Q-Q plot of the standardized residuals – each case's distance from its group's mean, or for a paired test each pair's difference from the mean difference – a look at normality as a plot rather than a test. `"levene"` or `"qq"` asks for one of them; `c("levene", "qq")` for both. Levene's test is not applicable when paired = TRUE. If NULL (default), defers to `joutput()`'s `diagnostics` setting, which is off at every output level until it is set. The Q-Q plot is also drawn by `jplot(result, which = "qq")`.
ci	Logical or NULL. If TRUE, adds 95% confidence interval for the mean difference. If NULL (default), defers to `joutput()`.

`subset` An optional unquoted logical expression (e.g. `Group == 1`) to subset cases for this call only. Applied after jcomplete and jsubset. Does not affect other function calls. Written in R syntax and checked the way `jsubset()` checks a filter: `subset = NOT(Age < 40)`, for example, is refused with the corrected `subset = !(Age < 40)` shown (see `jsubset` for a translation table).

`variable.id` Character or NULL. Variable label display mode: one of `"both"`, `"names"`, `"labels"`, `"legend"`, or `"legend.bottom"`. `"names"` shows variable names only; `"both"` shows `"name: label"`; `"labels"` shows the DV and grouping-variable labels in the table captions (group levels follow the value.id mode) – best for short labels; `"legend"`/`"legend.bottom"` keep names and print a label legend after the output. NULL (default) defers to `joutput()`'s `variable.id` setting. Not a logical.

`value.id` Character or NULL. Value-label display mode for the group descriptives rows: `"both"` (`"code: label"`), `"values"` (bare code), or `"labels"` (the label, degrading to the bare code where a code has none). `"legend"` and `"legend.bottom"` keep the bare code in the table and print a value-label legend after it (`"legend"` per-table, `"legend.bottom"` consolidated where multiple tables are produced). A no-op for grouping variables with no value labels. NULL (default) defers to `joutput()`'s `value.id` setting. Not a logical.

`case.processing.detail` Per-call override of the Case Processing Summary detail tier: one of `"none"`, `"totals"`, or `"per_code"`. `NULL` (default) uses the active `joutput()` level default. The Case Processing table itself prints only when a filter or listwise deletion excluded cases; otherwise a one-line N statement takes its place. See `?joutput` (`case.processing`).

`full` Logical. If TRUE, turns on effect.size and ci together. Does not override explicit FALSE values, and does not turn on diagnostics, which `diagnostics` alone governs.

`digits` Integer or NULL. Number of decimal places for continuous statistics in the output tables (range 0-7; `digits = 0` prints whole numbers with no trailing decimal point). Does not affect p-values, percentages, or integer quantities (counts, N, degrees of freedom), which keep their own fixed conventions. NULL (default) defers to `joutput()`'s `digits` setting (default 3).

`print` Logical or NULL. `FALSE` prints nothing – no table, no title, no note – and returns the result as every call does; errors and warnings still show. `TRUE` prints even under `joutput("silent")`. `NULL` (the default) prints unless the level is silent. For a script that keeps many results and wants no output: see `joutput`.

`...` Reserved for argument-name checking. Passing `levene`, the name of the diagnostics setting before version 0.9.219, produces an error that names `diagnostics`.

Details

A red title identifying the test type is printed first, followed by variable labels (if present), then the results tables.

A transformed outcome or grouping term in `formula` – `log(x)` and the like – is computed once on the analysis data and used by both the t-test and the group descriptives, so the two describe the

same values. The transforms supported inline, and those that must be created as a column first, are as documented for `jlm`. A value or vector from your workspace may be named inside a computed term, as in `lm()`: `I(x > cutoff)` with `cutoff <- 10`. A data frame may not: write `y ~ x` with `data = MyData`, not `MyData$y ~ MyData$x`. A variable may not: jstats analyzes the variables of a data frame, so a vector holding one value for each case – `w` in `I(x * w)`, or an object reached with `$` such as `lst$w` – is added to the data frame first (`MyData$w <- w`), where `lm()` would read it from the workspace; the stop gives that line. A vector of any other length used value by value is refused, where `lm()` would repeat the shorter one; a set used with `%in%` may have any length.

Value

Invisibly returns a list of class `jst_ttest` containing: `model` (the `t.test` result), `model_frame` (the analysis data frame used for plotting), `test_type`, `formula`, `descriptives`, `t`, `df`, `p`, `mean_difference`, `ci` (95% CI), `cohens_d`, `d_label`, `n`, `sample_info` (pipeline and missing data counts), and `levene`: Levene's test as printed, a list of `F`, `df1`, `df2`, `p` and `note` – the verdict of the note under a significant test (`"acceptable"`, `"unreliable"` or `"differ"`), recorded at every output level, or `NA` when no note applies (the test not significant, or Welch's test run); `NULL` when Levene's test was not asked for or the test is paired.

Unequal variances

Student's t-test assumes the two groups have the same variance, and Levene's test (`diagnostics = TRUE`) asks whether they do. When it is significant, a note under its table states the two things that decide how much that matters – the ratio of the larger group's size to the smaller, and of the larger standard deviation to the smaller – and then takes one of three forms. Textbooks give different guidelines and no single cutoff is agreed, so the note does not treat one as exact:

- Stevens (*Intermediate Statistics: A Modern Approach*; *Applied Multivariate Statistics for the Social Sciences*) holds that unequal variances distort the test appreciably only when the larger group is more than 1.5 times the smaller.
- Moore, McCabe and Craig (*Introduction to the Practice of Statistics*) treat results as approximately correct while the largest standard deviation is less than twice the smallest, a rule they state for the analysis of variance; Howell (*Statistical Methods for Psychology*) gives the same limit as a variance ratio of four and adds that unequal variances and unequal group sizes do not mix.

The note reads "usually still acceptable" when the larger group is no more than 1.25 times the smaller and the larger standard deviation no more than twice the smaller; it says the p-value may not be reliable when the group sizes differ by more than 1.5 times and the standard deviations by more than twice; and between the two it says that guidelines differ. The 1.25 is not a textbook figure. It comes from simulations run for jstats (a true null hypothesis, normal scores, a nominal 5 percent level, the smaller group the more variable): in the cases tried, Student's test rejected up to about 7 percent of the time inside the first range, up to about 10 percent in the middle one, and about 14 to 15 percent in the last. Two groups of the same size are the most forgiving case: the rate stayed near 5 percent with one standard deviation up to three times the other. The direction matters: when the LARGER group is the more variable, the test rejects too rarely instead. Welch's t-test (`welch = TRUE`) does not assume equal variances. The note is not printed at `joutput("minimal")`, which prints the table alone.

See Also

jstats for the package overview, workflow conventions, and complete function listing.

Examples

```
# With explicit data frame
jt(WellbeingScore ~ Volunteer, data = community)
jt(WellbeingScore ~ Volunteer, data = community, welch = TRUE)
jt(WellbeingScore ~ Volunteer, data = community, full = TRUE)

# The data frame may also be given positionally
jt(WellbeingScore ~ Volunteer, community)

# Checking the equal-variances assumption: Levene's test
jt(WellbeingScore ~ Volunteer, data = community, diagnostics = TRUE)

# Using juse() default
juse(community)
jt(WellbeingScore ~ Volunteer)
jt(WellbeingScore ~ Volunteer, full = TRUE)
```

---

jupdate *Update jstats to the latest version*

Description

jupdate() installs the most recent version of jstats. While jstats is in its pre-release phase this downloads and installs the latest pre-built version; once jstats reaches CRAN, the same command will update it the ordinary way. Either way, you run one command instead of having to remember an install line. It is safe to call from the console, a script, or a Quarto document.

Usage

```
jupdate(ask = FALSE, ...)
```

Arguments

ask	Logical. When TRUE and the session is interactive, jupdate() shows the available and installed versions and asks for confirmation before installing. Defaults to FALSE (update without prompting), which is also what happens in any non-interactive session, such as a Quarto render.
...	Not used. An input jupdate() does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Details

The function checks for an internet connection first; if jstats is already up to date it says so and stops. Otherwise it installs the update and then confirms that the update actually happened before reporting success.

Where the update goes. The update is installed into the library folder that holds the copy of jstats currently loaded, so the version in use is the one replaced. This matters when a computer has more than one library folder (a system-wide one and a personal one, say): a plain `install.packages("jstats")` puts the new version in whichever folder is first on `.libPaths()`, which can leave a second copy of jstats beside the original rather than replacing it. jupdate() follows the rule `update.packages()` uses instead, and installs where the loaded copy lives. To see where that is, run `find.package("jstats")`.

If the folder cannot be written to. Before installing, jupdate() checks that R can create files in that folder. If it cannot – typically a system-wide folder such as Program Files on Windows, or a folder locked by an institution's IT policy – jupdate() stops and names the folder, rather than installing a second copy somewhere else. Run R with permission to write there (on your own Windows computer, usually by starting RStudio as an administrator), or ask your IT department to install the update.

How success is confirmed. `install.packages()` only warns when it cannot download or install a package; it does not stop. jupdate() therefore does not take a quiet install as success. After the install it reads the version now on disk in the target folder, and reports success only when that version is newer than the one loaded. If it is not – a dropped connection part-way through, a repository the network cannot reach, or a failed build – jupdate() stops, states that the folder still holds the old version, and repeats what `install.packages()` reported about the cause so you can see why.

The separate process. The install runs in a separate R process so the copy of jstats loaded in your session does not lock its own files during the install (the usual cause of a failed update on Windows).

What is new. After a successful update jupdate() prints the news for every version newer than the one you had, read from the news file (NEWS.md) the new version installed: the newest version first, with the Markdown backticks and bold markers taken out, since the Console shows them as typed. When the file has no such entry, only the success message prints. The same notes are on the Changelog page of the package's reference site.

After the update. Restart R once to load the new version; the success message shows how.

Value

Invisibly `NULL`. Called for its side effect of installing the update, and for the messages it prints.

Examples

```
## Not run:
jupdate()            # update without prompting
jupdate(ask = TRUE)  # confirm before updating

## End(Not run)
```

`juse` *Set or display the default data frame for jstats functions*

Description

`juse()` sets a default data frame that will be used automatically by all jstats functions when the `data` argument is omitted. This reduces typing and makes interactive use more convenient.

The function stores the *name* of the data frame, not a copy of the data. This means any changes you make to the data frame (adding columns, recoding variables, etc.) are automatically reflected in subsequent function calls.

Usage

```
juse(data, ...)
```

Arguments

`data`	A data frame (unquoted). If omitted, prints the current default. Use `juse(NULL)` to clear the default.
`...`	Not used. An input `juse()` does not take – a misspelled argument – stops with "unused input" and the name as typed, in place of R's own "unused argument".

Value

Invisibly returns `NULL`. Called for its side effect of setting, displaying, or clearing the default data frame.

Note

`juse()` stores the *name* of the data frame, not a copy of the data. This means any changes you make to the data frame (adding columns, recoding variables, etc.) are automatically reflected in subsequent function calls. This differs from base R's `attach()`, which creates a snapshot that can become stale after modifications. `juse()` is the recommended approach for this package.

See Also

`jstats` for the package overview, workflow conventions, and complete function listing.

Examples

```
juse(community)             # Set community as the default
juse()                      # Display current default
jdesc(Age, WellbeingScore)  # Uses community automatically
juse(NULL)                  # Clear the default
```

Index

∗ **datasets**
clinic, 3
community, 4

clinic, 3, *5, 6*
community, *3*, 4, *4*

file.path, *30*

jai, 6
jalpha, 7
jaov, 9
javg, 13, *107*
jcomplete, 15
jconvert, 17, *35*, *72*, *74, 75*
jcopy, 21
jcorr, 22, *74*
jcount, 25, *49, 50*, *71, 72*, *85*, *98*
jcrosstab, 26
jdata_dir, 29
jdeclare_missing, 30, *45*, *48*, *63*, *92*
jdesc, 37
jdummy, *26*, 40, *49, 50*, *56*, *69*, *71, 72*, *85*, *98*
jencode, *18*, *25*, *32*, *34*, 42, *47*, *49*, *72*, *74*, *94*
jfreq, *45*, 46, *74*, *102*
jlikert, 49, *98*
jlm, *11*, *25*, 50, *111*
jload, *20*, *22*, *30*, 58, *74*, *90*, *98*
jlogistic, 65
jnumeric, *26*, *49, 50*, *54*, *68*, 71, *85*, *98*
joptions, *18*, *20*, *29, 30*, *32*, *34, 35*, *43*, *47*, *62, 63*, 72, *88*, *90*, *92*, *97, 98*
joutput, *8*, *11*, *24*, *28*, *39*, *47*, *54*, *67*, *72*, *74, 75*, 76, *102*, *110*
jplot, *49*, *56*, *69*, 82
jrecode, *35*, *45*, *74*, 87, *96*
jrelabel, *45*, *92*, 93
jsave, *22*, *30*, *49*, *61*, *74*, 97
jscreen, *25*, *49, 50*, 100
jstats, *9*, *12*, *14*, *16*, *24*, *29*, *35*, *39*, *41*, *48*, *57*, *64*, *70*, *75*, *81*, *86*, *92*, *96*, *99*, *103*, *106, 107*, *112*, *114*
jsubset, *8*, *10*, *23*, *27*, *37*, *46*, *51*, *65*, *83*, *101*, 104, *110*
jsum, *14*, 107
jt, 108
jupdate, 112
juse, *22*, *25*, *49*, *71*, 114