Package 'rmoriebricklayer'

September 17, 2026

Title Reproducible Data Capsules with Provenance and Fallback

Version 0.5.0

Description Tools for building brick-proof, reproducible, self-contained data capsules.
Resolves open-data sources through the Comprehensive Knowledge Archive Network ('CKAN', <https://ckan.org/>) package_show and package_search endpoints, records and verifies provenance with Secure Hash Algorithm 256 ('SHA-256') digests and Internet Archive 'Wayback Machine' (<https://web.archive.org/>) snapshots, validates downloaded data against a pinned schema, and falls back to schema-driven synthetic data when the real source is unreachable.
Run records are captured in a manifest plus a plain-language summary so any result can be traced back to its inputs. Distributional drift between a pinned capsule and a fresh fetch is tested with Kolmogorov-Smirnov, chi-square, population stability index, Jensen-Shannon divergence and 'Benford' first-digit screens, because a re-released extract can be statistically identical yet differ byte-for-byte, and a column can keep its name and type while having been silently rescaled. Manifests can be authenticated rather than only checksum-verified, with keyed digests ('HMAC-SHA-256', RFC 2104) or post-quantum hash-based signatures ('Winternitz' one-time signatures under a 'Merkle' tree, RFC 8391), and pinned chunk-wise through a 'Merkle' tree so a mismatch identifies which part of a capsule moved. Also ships a compiled C++ core (summary, robust and rank statistics, 'SHA-256', 'SHA-512' and 'CRC-32') that sibling packages in the 'rmorie' ecosystem reach through 'LinkingTo' for a single, shared numeric and provenance-hashing backend.
For the published administrative tables these capsules usually hold, it computes period-over-period change matched on the period rather than the row, with the exact conditional-binomial interval for a ratio of counts and with a percentage-point reading kept distinct from a percent change, rendered to Hypertext Markup Language ('HTML'), Portable Document Format ('PDF'), delimited text, JavaScript Object Notation ('JSON') or Markdown. Interval categories such as ``2 to 5'' or ``50+'' are parsed to bounds and the dependence of any derived figure on the open top band is measured rather than

assumed. Concentration is summarised by the 'Gini' coefficient, the Lorenz curve and tail-index estimation by exact discrete maximum likelihood; trend in a series of a few periods by the Mann-Kendall test with 'Theil-Sen' slopes, a permutation step-change scan and Poisson rate ratios; and region-coded counts by indirect standardisation, exact standardised incidence ratios, the empirical Bayes shrinkage of Clayton and 'Kaldor' (1987) <doi:10.2307/2532003>, funnel-plot limits and Moran's I.

License AGPL-3

Encoding UTF-8

Language en-US

Depends R (>= 4.1.0)

Imports methods, stats, utils

Suggests digest, jsonlite, knitr, openssl, pkgdown, rmarkdown, sf, stringi, testthat (>= 3.0.0)

VignetteBuilder knitr

URL https://github.com/rootcoder007/rmorie-bricklayer, https://rootcoder007.github.io/rmorie-bricklayer/

BugReports https://github.com/rootcoder007/rmorie-bricklayer/issues

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

RoxygenNote 7.3.3

SystemRequirements libcurl (deb: libcurl4-openssl-dev, rpm: libcurl-devel).

NeedsCompilation yes

Author Vansh Singh Ruhela [aut, cre] (ORCID: <https://orcid.org/0009-0004-1750-3592>)

Maintainer Vansh Singh Ruhela <vsruhela@proton.me>

Repository CRAN

Date/Publication 2026-09-16 22:40:08 UTC

Contents

`admissions` *Admissions implied by a population and a length of stay*

Description

Admissions implied by a population and a length of stay

Usage

```
admissions(adp, alos, t = 365)
```

Arguments

`adp`	Average daily population.
`alos`	Average length of stay in days.
`t`	Length of the period in days. Default 365.

Details

Rearranging the identity $\bar{X}_{hc} = N_a \bar{X}_t / t$ (Lakner 1976, p.18-20) for N_a. Useful when two of the three quantities are published and the third is not.

Value

The implied number of admissions.

See Also

`adp()`, `alos()`, `stock_flow()`

Examples

```
# Lakner (p.20) runs this the other way: t = 365, an average daily
# population of 25 and 1,750 admissions imply a stay of 5.2 days.
alos_implied <- 25 * 365 / 1750
round(alos_implied, 1)

# and back again
admissions(25, alos_implied)
```

---

adp *Average daily population*

Description

The mean number of person-days served per day over a period: a STOCK, answering how many people are held at one time.

Usage

```
adp(days, t = 365)
```

Arguments

`days`	Person-days served during the period. A vector is summed, so one element per person is the usual input.
`t`	Length of the period in days. Default 365.

Details

This is Lakner's $\bar{X}_{hc} = \sum X_i / t$ (1976, p.15). The denominator is TIME, which is what makes it a stock. Compare `alos()`, whose denominator is people.

Value

A single number: person-days per day.

References

Lakner, E. (1976) *A Manual of Statistical Sampling Methods for Corrections Planners*. University of Illinois at Urbana-Champaign.

See Also

`alos()`, `stock_flow()`, `adp_from_counts()`

Examples

```
# Lakner's own worked example (p.15): 3,000 inmates served 13,500
# detention days in a year.
adp(13500)

# per-person days give the same total
adp(c(10, 20, 30), t = 30)
```

`adp_from_counts` *Person-days estimated from periodic headcounts*

Description

When only a count of people present on certain days is available – not a record per person – the person-days over the period are the mean of those counts scaled to the period's length.

Usage

```
adp_from_counts(counts, t = 365)
```

Arguments

`counts`	Headcounts, one per day on which a count was taken.
`t`	Length of the period in days. Default 365.

Details

Lakner's $X_t = (\frac{1}{C}\sum N_i)t$ (1976, p.21). The counts need not cover every day: counting on weekdays only is the usual case, and the mean of the days counted stands in for the days not counted. That substitution is an assumption, and it fails if the days counted differ systematically from the days missed – weekday-only counting where weekend admissions are released before Monday, for instance.

Value

Estimated person-days over the period.

See Also

`adp()`

Examples

```
# Lakner (p.21): counts on 255 days summing to 34,935 imply just over
# fifty thousand detention days across the year.
adp_from_counts(rep(34935 / 255, 255))
```

agent_bundle *Agent-assisted reproducibility-bundle help*

Description

Forwards a bundle-building request to the `rmorie` command-line agent (optional binary from rmorie-cli), with a rmoriebricklayer-focused preamble. The agent can run R and read/write files to help assemble or repair a brick-proof bundle. See `rmorie::agent` for requirements.

Usage

```
agent_bundle(request, model = NULL, backend = "auto")
```

Arguments

`request`	Character scalar describing the bundle task.
`model`	Optional model id, e.g. `"minimax-m3:cloud"` for an Ollama server or `"claude-sonnet-5"` for Anthropic; defaults to `$RMORIE_AGENT_MODEL` or the CLI's auto-pick.
`backend`	`"auto"` (default), `"ollama"` (server from `$RMORIE_AGENT_OLLAMA_URL`) or `"anthropic"` (`$RMORIE_AGENT_API_KEY`) .

Value

Character scalar: the agent's output, or a message if the `rmorie` binary is not installed.

Examples

```
# Routed to the optional rmorie CLI agent when it is installed; with no
# binary on PATH each call returns an install hint instantly (no error,
# no network), so this is safe to execute anywhere.
agent_bundle("scaffold a bundle for analysis.R using the Toronto CKAN dataset")

# Free local/cloud route: an Ollama server (RMORIE_AGENT_OLLAMA_URL, default
# http://localhost:11434) serving the model the MORIE stack uses.
agent_bundle("add a Wayback fallback to my fetch step",
             backend = "ollama", model = "minimax-m3:cloud")

# Anthropic route (needs RMORIE_AGENT_API_KEY); the CLI's default model.
agent_bundle("repair the SHA256 provenance for my capsule",
             backend = "anthropic", model = "claude-sonnet-5")


# With no rmorie binary on PATH the call returns an install hint, not an
# error -- safe to run anywhere:
if (!nzchar(Sys.which("rmorie"))) agent_bundle("hello")
```

`alos` *Average length of stay*

Description

The mean number of person-days served per person: the FLOW side of the same person-days.

Usage

```
alos(days, n)
```

Arguments

`days`	Person-days served during the period, summed.
`n`	Number of people. For an unbiased average this should be the people both admitted AND released within the period, because anyone still held has an unfinished stay.

Details

Lakner's $\bar{X}_t = \sum X_i / N'$ (1976, p.16), with a caveat worth repeating (p.16-17): the period must be longer than the longest stay people actually serve, or the average is biased DOWNWARD, since the longest stays are the ones that fail to finish inside the window. For short-stay facilities a year is comfortable; for long sentences it is not, and the period has to be set from the records.

Value

A single number: days per person.

References

Lakner, E. (1976) *A Manual of Statistical Sampling Methods for Corrections Planners*. University of Illinois at Urbana-Champaign.

See Also

`adp()`, `stock_flow()`

Examples

```
# Lakner (p.17): 2,700 inmates admitted and released served 12,150
# detention days between them.
alos(12150, 2700)
```

apply_schema_validation *Apply Schema Validation, Stopping on Fatal Issues*

Description

Runs `validate_schema()` and acts on the result: fatal issues raise an error via `stop()`, warning-severity issues emit a `warning()`.

Usage

```
apply_schema_validation(df_raw, provenance)
```

Arguments

`df_raw`	The data frame to validate.
`provenance`	A provenance list as returned by `load_provenance()`.

Value

Invisibly, `TRUE` if no issues were found and `FALSE` otherwise. Errors if any fatal issue is present.

Examples

```
prov <- list(schema = list(expected_columns = "id"))
apply_schema_validation(data.frame(id = 1), prov)
# A missing required column is fatal:
try(apply_schema_validation(data.frame(x = 1), prov))
```

ascii_fallback *Use Text As-Is, Falling Back to ASCII When It Cannot Be Represented*

Description

Returns `x` unchanged when it is valid, well-formed text (so legitimate UTF-8 such as an accented name is preserved), and only transliterates to plain ASCII via `to_ascii()` when the text is not valid UTF-8 (an encoding error) or when `force = TRUE` (for ASCII-only destinations such as a package `DESCRIPTION`) . This lets author and supervisor names keep their accents wherever UTF-8 is supported while degrading gracefully instead of erroring where it is not.

Usage

```
ascii_fallback(x, force = FALSE)
```

Arguments

x	A character vector.
force	Logical; always transliterate to ASCII (default FALSE) .

Value

A character vector: x where it can be represented, ASCII otherwise.

Examples

```
# By default valid UTF-8 is preserved (accents kept where supported).
ascii_fallback("\u00c1ngela")              # "\u00c1ngela"

# force = TRUE always transliterates (for ASCII-only destinations
# such as a package DESCRIPTION).
ascii_fallback("\u00c1ngela", force = TRUE)  # "Angela"

# Plain ASCII is returned unchanged either way.
ascii_fallback("plain name")

# Vectorised; each element handled independently.
ascii_fallback(c("caf\u00e9", "resume"), force = TRUE)
```

---

band_sensitivity *How much a result depends on the open band's assumed cap*

Description

Recomputes a statistic across a range of assumed caps for the open top band and reports how far the answer moves. Everything derived from banded data carries this dependence; the only question is whether it was measured.

Usage

```
band_sensitivity(
  bands,
  counts,
  statistic = gini,
  caps = NULL,
  rule = "midpoint"
)

## S3 method for class 'rmbl_band_sensitivity'
print(x, ...)
```

Arguments

bands	A data frame from parse_bands(), or labels to parse.
counts	How many units fall in each band.
statistic	A function of a numeric vector of per-unit values. Defaults to gini().
caps	Caps to try for the open top band. Defaults to a geometric sweep from the band's lower bound to twenty times it.
rule	Passed to band_values().
x	An rmbl_band_sensitivity object.
...	Ignored.

Value

A data frame of cap and value, with the span and the relative span attached as attributes and printed by print().

Examples

```
b <- parse_bands(c("1", "2 to 5", "6 to 10", "Greater than 10"))
counts <- c(1200, 430, 110, 38)

s <- band_sensitivity(b, counts)
s

# A statistic that barely moves has been measured; one that swings
# has not.
band_sensitivity(b, counts, statistic = mean)
```

---

band_values *Representative values for banded categories*

Description

Turns bounds into the single number per band that a calculation needs, under a stated rule. The open band is the whole difficulty: it has no midpoint, so a cap has to be supplied or assumed, and the assumption is recorded in the result rather than absorbed into it.

Usage

```
band_values(
  bands,
  rule = c("midpoint", "lower", "upper", "geometric"),
  open_upper_cap = NULL,
  open_upper_factor = 2,
  open_lower_floor = 0
)
```

Arguments

bands	A data frame from parse_bands(), or labels to parse.
rule	How to place a value inside a closed band. "midpoint" is the arithmetic mean of the bounds. "lower" and "upper" give the conservative and generous readings, which together bracket whatever the truth is. "geometric" suits a quantity whose distribution inside the band is closer to log-uniform than uniform, which is usual for counts and durations.
open_upper_cap	Upper bound to assume for an open top band. The default multiplies the band's lower bound by open_upper_factor, which is an assumption and is flagged as one.
open_upper_factor	Multiplier used when no cap is given.
open_lower_floor	Lower bound to assume for an open bottom band. Defaults to zero.

Value

The band table with a value column and an assumed column marking the rows whose value rests on the open-band assumption.

See Also

band_sensitivity()

Examples

```
b <- parse_bands(c("1", "2 to 5", "6 to 10", "Greater than 10"))

band_values(b)

# The lower and upper rules bracket the truth.
band_values(b, rule = "lower")$value
band_values(b, rule = "upper", open_upper_cap = 40)$value
```

---

benford_test *Benford first-digit test*

Description

Compares the distribution of leading significant digits in x with Benford's law, $P(d) = \log_{10}(1 + 1/d)$. Naturally occurring quantities that span several orders of magnitude follow it closely; figures that were rounded, truncated, capped, re-scaled, or invented typically do not. That makes it a cheap screen for a numeric column that arrived looking plausible but is not the measurement it claims to be.

Usage

```
benford_test(x)
```

Arguments

x	Numeric vector.

Details

It is a SCREEN, not a verdict. Columns with a narrow range, a unit floor or ceiling, or an assigned-identifier structure (postcodes, year fields, prices ending in 99) legitimately violate Benford's law. Treat a small p-value as a reason to look, never as evidence of fabrication.

Zeros and non-finite values have no leading significant digit and are excluded; the sign is ignored.

Value

A list of class `bricklayer_benford`: `counts` (observed digit frequencies 1–9), `expected`, `proportion`, `statistic`, `df`, `p_value`, and `n`.

References

Benford F (1938). The law of anomalous numbers. *Proceedings of the American Philosophical Society* 78(4), 551–572.

Examples

```
# A quantity spanning several orders of magnitude follows the law.
set.seed(3)
benford_test(10^stats::runif(2000, 0, 6))

# Digits drawn uniformly do not.
benford_test(as.numeric(paste0(sample(1:9, 2000, TRUE), "000")))

# The expected proportions are the closed form.
b <- benford_test(10^stats::runif(500, 0, 5))
all.equal(b$expected / b$n, log10(1 + 1 / (1:9)))

# Zeros carry no leading digit and are excluded from n.
benford_test(c(0, 0, 1, 2, 3))$n
```

`bricklayer_fetch` *Fetch a URL to disk with an Internet Archive fallback (C++/libcurl)*

Description

The shared data-fetch foundation of the morie ecosystem. Downloads `url` to `dest`; if the live download fails (404, network), it retries the Wayback Machine snapshot – so a rotated or removed source file (CIHI, open-data portals) stays retrievable. Backed by the package's C++ `rmbl_fetch_with_fallback` kernel (libcurl), which **rmorie** and **morie** share through `LinkingTo`.

Usage

```
bricklayer_fetch(url, dest, wayback = "", timeout = 120L)
```

Arguments

url	Live source URL.
dest	Destination file path.
wayback	Optional explicit Wayback snapshot URL. "" (default) auto-resolves one via wayback_snapshot_url.
timeout	Per-request timeout, seconds.

Value

Invisibly, one of "live", "wayback", or throws on total failure.

Examples

```
# Inputs are validated before any network access:
try(bricklayer_fetch("", tempfile()))          # empty url -> error


# Downloads from the live web service; try() keeps the example graceful
# when neither the live URL nor its Wayback fallback is reachable.
dst <- tempfile(fileext = ".xlsx")

# Live download; auto-resolves a Wayback snapshot only if the live URL fails.
try(bricklayer_fetch(
  paste0("https://www.cihi.ca/sites/default/files/document/",
         "hospital-beds-2024-2025-data-tables-en.xlsx"),
  dst))

# Pin an explicit Wayback snapshot to fall back to, and a shorter timeout.
try(bricklayer_fetch(
  "https://example.org/rotated-file.csv", tempfile(fileext = ".csv"),
  wayback = "https://web.archive.org/web/2024id_/https://example.org/rotated-file.csv",
  timeout = 60))

# The return value tells you which source served the file.
status <- try(bricklayer_fetch("https://cloud.r-project.org/", tempfile()))
status   # "live" or "wayback"
```

---

bricklayer_fetch_parse_siu

Fetch and parse one SIU director's report

Description

Convenience: bricklayer_fetch_siu() then bricklayer_parse_siu(). Fails gracefully – returns NULL with a message when the report cannot be retrieved.

Usage

```
bricklayer_fetch_parse_siu(drid, lang = c("en", "fr"))
```

Arguments

drid	Director's-report id (the drid= query parameter).
lang	"en" (default) or "fr".

Value

A named character vector of parsed fields, or NULL when the fetch fails.

Examples

```
f <- try(bricklayer_fetch_parse_siu(648), silent = TRUE)
if (is.character(f)) f[["police_service"]]
```

---

bricklayer_fetch_siu *Fetch an Ontario SIU director's report by drid*

Description

Downloads the HTML of one Ontario Special Investigations Unit (SIU) director's report to dest, via bricklayer_fetch (live URL with a Wayback Machine fallback). This is the fetch step of the open SIU corpus pipeline: pair it with the SIU parser in **rmorie** (or the standalone siu C++ package) to rebuild the full director's-report corpus yourself, then audit it with the multi-agent panel.

Usage

```
bricklayer_fetch_siu(
  drid,
  dest,
  lang = c("en", "fr"),
  wayback = "",
  timeout = 120L
)
```

Arguments

drid	Director's-report id – the drid= query parameter (integer or integer-like scalar).
dest	Destination file path for the fetched HTML.
lang	"en" (default) or "fr".
wayback	Optional explicit Wayback snapshot URL (passed through to bricklayer_fetch) ; "" lets it discover one.
timeout	Request timeout in seconds. Default 120.

Value

"live" or "wayback" (invisibly), as bricklayer_fetch.

See Also

bricklayer_fetch

Examples

```
# Downloads from the live SIU web service; try() keeps the example
# graceful when the service (and its Wayback fallback) is unreachable.
# Fetch report drid 648 to a temp file.
dest <- tempfile(fileext = ".html")
try(bricklayer_fetch_siu(648, dest))
```

---

bricklayer_json_from_json

Parse JSON into R objects (jsonlite's fromJSON, natively)

Description

Parse JSON into R objects (jsonlite's fromJSON, natively)

Usage

```
bricklayer_json_from_json(
  txt,
  simplifyVector = TRUE,
  simplifyDataFrame = simplifyVector,
  simplifyMatrix = simplifyVector,
  flatten = FALSE,
  bigint_as_char = FALSE,
  simplify = NULL,
  ...
)
```

Arguments

txt	JSON text, a file path, or an http(s) URL.
simplifyVector, simplifyDataFrame, simplifyMatrix, flatten	as in jsonlite.
bigint_as_char	integers beyond 2^53 come back as strings.
simplify	legacy: FALSE turns every simplification off.
...	ignored, for call compatibility.

Value

an R object.

Examples

```
bricklayer_json_from_json('[{"a":1,"b":"x"},{"a":2,"b":"y"}]')
bricklayer_json_from_json('[[1,2],[3,4]]')
```

`bricklayer_json_to_json` *Encode an R object as JSON (jsonlite's toJSON, natively)*

Description

Every option of jsonlite's `toJSON()` with the same default and the same bytes out: `dataframe`, `matrix`, `Date`, `POSIXt`, `factor`, `complex`, `raw`, `null`, `na`, `auto_unbox`, `digits` (`I(n)` for significant digits, `NA` for 15), `pretty` (TRUE = 2 spaces, or a width), `force`, plus `rownames`, `keep_vec_names`, `json_verbatim`, `always_decimal`, `time_format`, `UTC`, `no_dots`, `hms`.

Usage

```
bricklayer_json_to_json(
  x,
  dataframe = c("rows", "columns", "values"),
  matrix = c("rowmajor", "columnmajor"),
  Date = c("ISO8601", "epoch"),
  POSIXt = c("string", "ISO8601", "epoch", "mongo"),
  factor = c("string", "integer"),
  complex = c("string", "list"),
  raw = c("base64", "hex", "mongo", "int", "js"),
  null = c("list", "null"),
  na = c("null", "string"),
  auto_unbox = FALSE,
  digits = 4,
  pretty = FALSE,
  force = FALSE,
  ...
)
```

Arguments

`x`	the object to encode.
`dataframe`	As in jsonlite.
`matrix`	As in jsonlite.
`Date`	As in jsonlite.
`POSIXt`	As in jsonlite.

factor	As in jsonlite.
complex	As in jsonlite.
raw	As in jsonlite.
null	As in jsonlite.
na	As in jsonlite.
auto_unbox	As in jsonlite.
digits	As in jsonlite.
pretty	As in jsonlite.
force	As in jsonlite.
...	As in jsonlite.

Value

a length-one character vector of class `json`.

Examples

```
bricklayer_json_to_json(list(a = 1:3, b = "x"), auto_unbox = TRUE)
bricklayer_json_to_json(data.frame(id = 1:2, v = c(1.5, NA)), pretty = TRUE)
```

---

bricklayer_parse_siu *Parse an SIU director's report into the schema fields*

Description

Deterministic, offline extraction of every `bricklayer_siu_schema()` field (plus `_language`) from report HTML. Fields the report does not state come back as `""`.

Usage

```
bricklayer_parse_siu(html)
```

Arguments

html	A length-1 character vector of raw report HTML, or the path to a saved report file (e.g. from `bricklayer_fetch_siu()`) .

Value

A named character vector: the 16 schema fields plus `_language`.

Examples

```
f <- bricklayer_parse_siu(system.file("extdata",
                                      "siu_synthetic_report.html",
                                      package = "rmoriebricklayer"))
f[["number_of_subject_officers"]]
```

bricklayer_siu_iso_date

Convert a human-readable SIU report date to ISO format

Description

`"January 5, 2023"` (or `"January 5 2023"`) becomes `"2023-01-05"`; unparseable input becomes `""`.

Usage

```
bricklayer_siu_iso_date(x)
```

Arguments

`x`	A character vector of human-readable dates.

Value

A character vector of `YYYY-MM-DD` strings (or `""`) .

Examples

```
bricklayer_siu_iso_date(c("January 5, 2023", "not a date"))
```

bricklayer_siu_resolve_so

Resolve the subject-official count from SIU report text

Description

Deterministic, reproducible extraction of the subject-official (SO) count for reports where a model panel (or a human) is unsure. The standard SIU privacy boilerplate is stripped first, then rules apply most-specific first: highest `SO #N` ordinal; spelled-out plural; singular subject official present (1); witness-official-only (0, a real answer); otherwise unresolved (`NA`) .

Usage

```
bricklayer_siu_resolve_so(text)
```

Arguments

`text`	A length-1 character vector of plain report text (see `bricklayer_siu_text()`) .

Details

bricklayer is the foundation layer: this function is the pure rule set. Reports already in the panel-reviewed corpus should never be re-derived – use `rmorie::morie_siu_resolve_so()`, which returns the verified corpus value first and only falls back to these rules for unreviewed reports.

Value

A list with `count` (integer, `NA` when unresolved) and `reason` (the human-readable evidence).

Examples

```
bricklayer_siu_resolve_so(
  "Subject Officials\nSO #1 Interviewed\nSO #2 Declined interview")
```

---

bricklayer_siu_schema *The panel-reviewed SIU report field schema*

Description

The sixteen fields extracted from every Special Investigations Unit director's report. Count-type fields (`is_count = TRUE`) count distinct entities and zero is a real answer – a witness-official-only investigation has zero subject officials.

Usage

```
bricklayer_siu_schema()
```

Value

A data.frame with columns `name`, `is_count`, and `description`.

Examples

```
bricklayer_siu_schema()
```

bricklayer_siu_text *Convert SIU report HTML to plain text*

Description

Convert SIU report HTML to plain text

Usage

```
bricklayer_siu_text(html)
```

Arguments

html	A length-1 character vector of raw report HTML.

Value

A length-1 character vector of plain text.

Examples

```
bricklayer_siu_text("<p>Number of SIU Investigators assigned: 3</p>")
```

capsule_attest *Attest a capsule, binding a signature to what it covers*

Description

Signs a manifest's canonical digest and records everything a verifier needs alongside it: the scheme, the public key, the context string, the digest that was signed, and the manifest's seal if it came from a chain. capsule_check_attestation() checks the result against a manifest without being told anything further.

Usage

```
capsule_attest(manifest, key, context = NULL, prehash = "none", note = NULL)

capsule_check_attestation(attestation, manifest, key_expected = NULL)
```

Arguments

manifest	A manifest, as from make_manifest().
key	A signing key from fips_keygen() or pqc_keygen().
context	Optional context string, bound into the signature for the standardised schemes; see capsule_sign().
prehash	Pre-hash, as in capsule_sign().
note	Optional free text recorded in the attestation – what the signature is meant to assert, in the signer's own words. It is covered by the signature, since it is part of the digest.
attestation	An attestation from capsule_attest().
key_expected	Optional public key hex the attestation must carry. Supply it when you know which key should have signed: without it the check confirms the attestation is internally consistent, which any key's holder could arrange.

Details

What this adds over capsule_sign(). A bare signature leaves three things implicit – which key, which scheme, and which bytes. A verifier who has to be told those out of band cannot check anything they were not already given, which makes the signature a formality. An attestation states them, so the check is capsule_check_attestation(attestation, manifest) and nothing else.

What it does NOT establish. That the public key belongs to whoever you think it does: an attestation is only as good as the channel the key arrived on. And that the manifest is true – only that it has not changed since it was signed. capsule_falsify() is for the other question.

Value

capsule_attest() a list of class bricklayer_attestation; capsule_check_attestation() a list with ok and a checks data frame, one row per check.

See Also

capsule_sign(), manifest_digest(), capsule_falsify(), chain_seal().

Examples

```
m <- make_manifest(list(dataset = "otis", rows = 1200L),
                   environment = FALSE)
key <- fips_keygen("ML-DSA-65")
att <- capsule_attest(m, key, note = "counts as published")

# a verifier needs the attestation and the manifest, nothing else
res <- capsule_check_attestation(att, m)
res$ok
res$checks

# any change to the manifest breaks it
m2 <- m
m2$meta$rows <- 1201L
```

```
capsule_check_attestation(att, m2)$ok

# and so does presenting a different key
capsule_check_attestation(att, m,
  key_expected = fips_keygen("ML-DSA-65")$public)$ok
```

`capsule_bundle` *Bundle a capsule into one signed artifact*

Description

Records a digest of every file named, the manifest's canonical digest, and an attestation covering both, as a single JSON file. `capsule_bundle_verify()` re-hashes the files on disk and checks everything against it.

Usage

```
capsule_bundle(
  dir,
  manifest,
  key,
  files = NULL,
  context = NULL,
  prehash = "none",
  note = NULL,
  path = NULL
)

capsule_bundle_read(path)

capsule_bundle_verify(bundle, dir, manifest = NULL, key_expected = NULL)
```

Arguments

`dir`	Directory the capsule lives in.
`manifest`	A manifest, as from `make_manifest()`.
`key`	A signing key from `fips_keygen()` or `pqc_keygen()`.
`files`	Paths relative to `dir`. Defaults to every regular file in `dir`, excluding the bundle itself.
`context`, `prehash`, `note`	As in `capsule_attest()`.
`path`	Where to write the bundle. Defaults to `capsule_bundle.json` inside `dir`.
`bundle`	A bundle read back with `capsule_bundle_read()`, or the path to one.
`key_expected`	Optional public key hex the bundle's attestation must carry. Supply it when you know which key should have signed: without it the check confirms the bundle is internally consistent, which any key's holder could arrange.

Details

The file digests are inside the signed payload, not alongside it. That is the whole point: a list of hashes that is not itself signed can be rewritten to match whatever the files now say, and a recipient who re-hashes the files and compares them to that list learns only that the list is consistent with itself.

What it establishes: the named files have not changed since signing, the manifest has not changed, and both were signed together by the holder of that key. What it does not: that the key belongs to anyone in particular, or that the manifest's claims are true.

Value

`capsule_bundle()` the bundle, invisibly, with the path it was written to as an attribute; `capsule_bundle_read()` the bundle; `capsule_bundle_verify()` a list with `ok` and a `checks` data frame.

See Also

`capsule_attest()`, `verify_capsule()`, `manifest_digest()`.

Examples

```
dir <- tempfile()
dir.create(dir)
write.csv(data.frame(x = 1:3), file.path(dir, "data.csv"),
          row.names = FALSE)
m <- make_manifest(list(dataset = "demo"), environment = FALSE)
key <- fips_keygen("ML-DSA-44")

b <- capsule_bundle(dir, m, key, note = "as published")
capsule_bundle_verify(attr(b, "path"), dir, manifest = m)$ok

# touch a byte of the data and the bundle no longer holds
write.csv(data.frame(x = 1:4), file.path(dir, "data.csv"),
          row.names = FALSE)
capsule_bundle_verify(attr(b, "path"), dir, manifest = m)$ok
unlink(dir, recursive = TRUE)
```

---

capsule_drift *Compare a fetched data frame with the one a capsule was pinned against*

Description

Runs the appropriate drift test on every shared column and collects the verdicts in one report: `drift_ks()` plus `drift_psi()` for a numeric column, `drift_homogeneity()` for a categorical one. Columns that appeared or vanished are listed separately, since no test applies to them.

Usage

```
capsule_drift(
  reference,
  current,
  alpha = 0.01,
  psi_threshold = 0.25,
  psi_min_n = 1000L,
  bins = 10L
)
```

Arguments

reference	Data frame the capsule was built from.
current	Data frame just fetched.
alpha	Significance level for the drifted flag (default 0.01; deliberately stricter than 0.05 because a wide table runs many tests).
psi_threshold	PSI above which a numeric column is flagged even when its p-value is not significant (default 0.25, the conventional "material shift" line).
psi_min_n	Minimum size BOTH samples must reach before psi_threshold is allowed to flag a column on its own (default 1000). The PSI bands are large-sample heuristics with no calibrated null distribution: on a few hundred rows, binning noise alone routinely pushes the index past 0.25, so applying the threshold there manufactures drift. Below this size the flag rests on the Kolmogorov-Smirnov p-value, which is calibrated, and the PSI is still reported as an effect size.
bins	Bins passed to drift_psi() (default 10).

Details

The categorical test is the two-sample homogeneity test, NOT drift_chisq() 's goodness-of-fit against a known distribution: the reference here is itself a finite sample, and ignoring its sampling error would report drift too readily.

This is the check that a byte-level digest cannot make. A re-released extract legitimately has a different SHA-256 while being the same data statistically; conversely a column can keep its name, type and row count while having been silently rescaled. capsule_drift() asks whether the DATA moved.

Value

A list of class bricklayer_drift: columns (a data frame, one row per shared column, with column, type, statistic, p_value, psi, js_divergence and drifted) , added, removed, n_reference, n_current, alpha, and any_drift.

See Also

validate_schema() for the structural check, which this complements rather than replaces.

Examples

```
set.seed(5)
ref <- data.frame(
  value = stats::rnorm(300),
  size = stats::runif(300, 1, 10),
  grade = sample(c("a", "b", "c"), 300, TRUE)
)

# A fresh draw from the same process: no drift.
same <- data.frame(
  value = stats::rnorm(300),
  size = stats::runif(300, 1, 10),
  grade = sample(c("a", "b", "c"), 300, TRUE)
)
d <- capsule_drift(ref, same)
d$any_drift

# A silently rescaled column, and a new category, are both caught.
moved <- same
moved$size <- moved$size * 3
moved$grade[1:100] <- "z"
capsule_drift(ref, moved)

# The PSI is always reported as an effect size, but on a sample this
# small it is not allowed to raise the flag by itself -- binning noise
# alone would clear 0.25. Lower psi_min_n to override that.
capsule_drift(ref, same)$columns$psi

# Structural changes are reported rather than tested.
capsule_drift(ref, same[, c("value", "grade")])$removed
```

---

capsule_falsify *Run falsification controls against a statistic*

Description

Recomputes `statistic` under conditions in which its value is known in advance, and reports whether it behaved. Four controls, each answering a different way of being wrong:

Usage

```
capsule_falsify(
  data,
  statistic,
  treatment = NULL,
  n = 199L,
  subset_frac = 0.8,
  seed = NULL
)
```

Arguments

`data`	A data frame.
`statistic`	A function of one data frame returning a single finite number.
`treatment`	Name of the column the finding is about, permuted for the permutation and placebo controls. Optional: without it those two controls are skipped and reported as such rather than silently omitted.
`n`	Number of permutations and subsets. The permutation p-value cannot be smaller than `1 / (n + 1)`, which is reported.
`subset_frac`	Fraction of rows kept by the subset control.
`seed`	Optional integer seed. Supplied, the result is reproducible; omitted, the current RNG state is used and recorded.

Details

- **permutation** – shuffle `treatment` and the association it carries is destroyed, so the statistic should fall to its null distribution. Reports where the observed value sits in that distribution, and the smallest p-value the number of permutations could have produced.
- **random common cause** – add a column of noise. It cannot possibly matter, so a statistic that moves is reading something other than the data.
- **placebo treatment** – replace `treatment` with a random draw of the same shape. The effect should vanish; if it does not, the statistic is picking up structure that has nothing to do with the exposure.
- **subset stability** – recompute on random subsets. Wide spread means the finding rests on particular rows, which is worth knowing before it rests on a conclusion.

What a pass means. Only that these controls did not catch anything. They are falsification tests, so they can refute and cannot confirm: passing all four is consistent with a statistic that is wrong for a reason none of them probes.

Value

A list of class `bricklayer_falsification`: `observed`, a `controls` data frame (one row per control, with `passed`) , and `permutation` holding the null distribution.

See Also

`capsule_power()` for the positive control – whether a real effect would have been seen at all; `capsule_attest()` for the different question of whether the record is intact rather than whether the finding survives.

Examples

```
set.seed(1)
d <- data.frame(x = rnorm(200))
d$y <- 0.8 * d$x + rnorm(200)
# a real association survives its controls
real <- capsule_falsify(d, function(z) cor(z$x, z$y),
```

```
                        treatment = "x", n = 199, seed = 42)
real$controls[, c("control", "passed")]

# and a statistic that ignores the data fails the ones that can see it
fake <- capsule_falsify(d, function(z) 0.5, treatment = "x",
                        n = 199, seed = 42)
fake$controls[fake$controls$control == "permutation", "passed"]
```

`capsule_power` *Detect an injected effect of known size*

Description

Adds an effect of each given size to the data, reruns the whole detection procedure, and reports how often it was found. The result is a power curve, and the smallest size detected reliably is the smallest effect the analysis could have seen.

Usage

```
capsule_power(
  data,
  statistic,
  inject,
  sizes = c(0, 0.2, 0.5),
  treatment = NULL,
  n = 99L,
  reps = 10L,
  alpha = 0.05,
  seed = NULL
)
```

Arguments

`data`	A data frame.
`statistic`	A function of one data frame returning a single finite number.
`inject`	A function of `(data, size)` returning the data with an effect of that size added. It is the caller's, because what counts as an effect of size 0.2 is a modelling decision and not something this function can guess.
`sizes`	Effect sizes to try. `0` is added if absent, since the detection rate there is the false-positive rate and belongs on the same curve.
`treatment`	Column permuted to build the null, as in `capsule_falsify()`.
`n`	Permutations per test.
`reps`	Repetitions per size. The detection rate at each size is out of this many, so its resolution is `1 / reps`.
`alpha`	Significance threshold for counting a detection.
`seed`	Optional integer seed.

Details

This is the positive control that capsule_falsify() lacks: its four are all negative. Those establish that the finding is not an artefact; this establishes that a real effect would not have been missed. A study that passes every falsification control and has no power against the effect it was looking for has not found that the effect is absent – it has found nothing either way, and the two are routinely reported as the same thing.

Value

A list of class bricklayer_power: a curve data frame (size, detected, reps, rate) , the alpha used, and smallest_detected, the smallest size found at a rate of at least 0.8 – NA when no size reached it.

See Also

capsule_falsify() for the negative controls.

Examples

```
set.seed(1)
d <- data.frame(x = rnorm(120), y = rnorm(120))
# inject a linear effect of x on y
inj <- function(z, size) {
  z$y <- z$y + size * z$x
  z
}
pw <- capsule_power(d, function(z) cor(z$x, z$y), inj,
                    sizes = c(0, 0.3, 0.6), treatment = "x",
                    n = 99, reps = 5, seed = 42)
pw$curve

# The rate at size 0 estimates the false-positive rate. With few
# repetitions it is usually 0, because alpha is small -- five draws
# at 0.05 come up empty about three times in four -- so read it as a
# sanity check that it is not LARGE, not as an estimate of alpha.
pw$curve[pw$curve$size == 0, "rate"]
```

---

capsule_report *Assess a capsule in one call*

Description

Runs the checks this package provides over one data frame and collects the findings into a single report: the structural schema check, the distributional comparison against a reference, the missingness picture, the multivariate outliers, a Benford screen on the wide numeric columns, and the integrity digests.

Usage

```
capsule_report(
  data,
  reference = NULL,
  schema = NULL,
  rules = NULL,
  chunks = NULL,
  signature = NULL,
  key = NULL,
  alpha = 0.01,
  max_rows_outliers = 20000L
)
```

Arguments

data	The data frame to assess.
reference	Optional data frame the capsule was pinned against. Supplying it enables the drift comparison, which is the check a digest cannot make.
schema	Optional schema from infer_schema(), or a provenance list containing one. Supplying it enables the structural check.
rules	Optional list of rule() objects.
chunks	Optional character vector of capsule chunks (see chunk_file()) to pin with a Merkle root.
signature, key	Optional signature and verifying key, as from capsule_sign(), checked against the data's own digest.
alpha	Significance level passed to capsule_drift().
max_rows_outliers	Skip the outlier scan above this many rows (default 20000), since it factors a covariance per call.

Details

Nothing here is new arithmetic. The value is that the answers arrive together and ordered by severity, because the failure mode this is built against is a person running one check, seeing it pass, and concluding the data is fine.

Value

A list of class bricklayer_report: findings (a data frame of severity, check, subject, detail) , verdict ("fatal", "warn", "note" or "clean") , profile, missingness, drift, digest, and n_rows / n_cols.

What "severity" means

- fatal – a required column is missing. Nothing downstream can run.
- warn – something moved: a column drifted, a value left its pinned range, missingness rose, a signature did not verify.

- `note` – worth a look but not necessarily wrong: outliers, a Benford departure, a constant column.

A clean report is not proof the data is correct. It means these particular checks found nothing, and every one of them has a stated blind spot – see `capsule_drift()` on statistical power and `benford_test()` on why a departure is a screen rather than a verdict.

See Also

`capsule_drift()`, `profile_columns()`, `validate_schema()`, `report_markdown()` to write it out.

Examples

```
set.seed(1)
ref <- data.frame(
  id = 1:200,
  score = stats::runif(200, 0, 10),
  grade = sample(c("a", "b", "c"), 200, TRUE),
  stringsAsFactors = FALSE
)

# A fresh extract from the same process: nothing to report.
cur <- ref
cur$score <- stats::runif(200, 0, 10)
capsule_report(cur, reference = ref, schema = infer_schema(ref))

# One rescaled column and a new category: both surface, worst first.
bad <- cur
bad$score <- bad$score * 5
bad$grade[1:80] <- "z"
r <- capsule_report(bad, reference = ref, schema = infer_schema(ref))
r
r$verdict
r$findings[, c("severity", "check", "subject")]

# A missing column is fatal, because nothing downstream can run.
capsule_report(bad[, c("id", "score")], reference = ref,
               schema = infer_schema(ref))$verdict
```

`capsule_sign` *Sign a capsule manifest*

Description

Authenticates `message` – normally a manifest digest, or the whole manifest text – so a verifier can tell that it came from the holder of the key and has not been altered since.

Usage

```
capsule_sign(
  message,
  key,
  scheme = NULL,
  context = NULL,
  deterministic = FALSE,
  prehash = "none"
)
```

Arguments

message	Length-1 character vector (or raw vector) to sign.
key	A shared secret (character/raw) for "hmac", a bricklayer_signing_key from pqc_keygen() for "xmss", or a bricklayer_fips_key from fips_keygen() for a standardised scheme (in which case scheme is taken from the key and ignored).
scheme	"xmss" (post-quantum, asymmetric) or "hmac" (symmetric). Inferred from key when not given.
context	Optional context string (character or raw, at most 255 bytes) for the standardised schemes, and ignored by the others. FIPS 204 and FIPS 205 both bind it into the message encoding, so a signature made under one context does not verify under another – which is how the same key is safely used for two purposes.
deterministic	For the standardised schemes, use the deterministic variant rather than drawing fresh randomness per signature. The signature then depends only on the key, message and context, which is what the standards' test vectors rely on.
prehash	For the standardised schemes, sign a digest of the message rather than the message itself – HashML-DSA (FIPS 204 section 5.4) or HashSLH-DSA (FIPS 205 section 10.2.2). One of "none" (the default, the pure variants), "sha256", "sha512", "shake128" or "shake256". The identifier of the pre-hash is bound into the signature, so a pre-hashed signature is never interchangeable with a pure one over the same digest.

Details

With scheme = "hmac" the key is a shared secret string and the result is an HMAC-SHA-256 tag. Symmetric, so anyone who can verify can also sign.

With scheme = "xmss" the key is a pqc_keygen() object and the result is a post-quantum one-time signature under the key's Merkle root. Asymmetric: a verifier holding only the public root cannot forge.

Value

A list of class bricklayer_signature: scheme, signature, and for XMSS also auth, index, root, height, key_state, randomizer (the per-signature R of RFC 8391, which a verifier needs and which therefore travels with the signature) and wire – the signature in the RFC's own byte order, index || R || WOTS || auth, hex-encoded. wire is byte-identical to what the XMSS reference

implementation produces from the same key material, so it can be handed to another implementation as bytes.

The returned key state must be carried forward

An XMSS signature consumes a leaf. The returned object therefore carries `key_state`, the key with `next_index` advanced, and **subsequent signing must use that** – reusing an index breaks the scheme. Passing an exhausted key is an error, not a silent wrap-around.

See Also

`capsule_verify()`, `core_hmac_sha256()`

Examples

```
# Symmetric: one shared secret.
sig <- capsule_sign("sha256:abc123", key = "shared-secret",
                    scheme = "hmac")
capsule_verify("sha256:abc123", sig, "shared-secret")
capsule_verify("sha256:TAMPERED", sig, "shared-secret")

# Post-quantum: the verifier needs only the public root.
key <- pqc_keygen(height = 2)
s1 <- capsule_sign("manifest-1", key)
capsule_verify("manifest-1", s1, signing_public_key(key))

# Carry the advanced key state forward for the next signature.
key <- s1$key_state
key$next_index
s2 <- capsule_sign("manifest-2", key)
capsule_verify("manifest-2", s2, signing_public_key(key))

# A signature does not transfer to another message.
capsule_verify("manifest-1", s2, signing_public_key(key))
```

`capsule_verify` *Verify a capsule manifest signature*

Description

Checks `signature` against `message`. For `"hmac"` the comparison is constant-time. For XMSS the Winternitz chains are walked to their ends and the authentication path replayed to the Merkle root; the digest is bound to both the leaf index and the root, so a signature cannot be replayed at another index or under another key.

Usage

```
capsule_verify(message, signature, key, context = NULL, prehash = NULL)
```

Arguments

message	The message the signature is claimed to cover.
signature	A bricklayer_signature from capsule_sign().
key	The shared secret for "hmac", a public key (or full signing key) for XMSS, or a fips_keygen() key or its fips_public_key() for a standardised scheme. A signature is not verified against a key of a different scheme.
context	The context string the signature was made under, for the standardised schemes. A signature made under a different context, or under none, does not verify.
prehash	The pre-hash the signature was made with. Taken from the signature when not given, since capsule_sign() records it; supply it to check a signature that arrived without that field.

Details

Returns FALSE rather than erroring on a malformed or truncated signature: a verifier must treat unparseable input as "not verified", never as an exception to be caught and ignored.

Value

A length-1 logical.

Examples

```
key <- pqc_keygen(height = 2)
sig <- capsule_sign("pinned-manifest", key)
pub <- signing_public_key(key)

capsule_verify("pinned-manifest", sig, pub)

# Every way of being wrong returns FALSE.
capsule_verify("edited-manifest", sig, pub)              # message changed
bad <- sig; bad$signature <- paste0("ff", substring(bad$signature, 3))
capsule_verify("pinned-manifest", bad, pub)              # signature edited
capsule_verify("pinned-manifest", sig,
               signing_public_key(pqc_keygen(height = 2)))  # foreign key

# A truncated signature is not verified, and does not error.
trunc <- sig; trunc$signature <- substring(sig$signature, 1, 64)
capsule_verify("pinned-manifest", trunc, pub)
```

---

capture_dependencies *Where each loaded package came from*

Description

Records, for every package named, its version and enough about its provenance to find the same one again: the library it was loaded from, the repository it was installed from, and – for a package installed from a remote – the remote's URL and commit.

Usage

```
capture_dependencies(packages = loadedNamespaces())
```

Arguments

packages	Character vector of package names. Defaults to the namespaces currently loaded.

Details

Why versions alone are not enough. Two installations can report the same version and differ: one built from CRAN, one from a fork, one from a local `R CMD INSTALL` of a working tree with uncommitted changes. A version number identifies an intention; the repository and commit identify what was actually loaded.

Value

A data frame with one row per package: `package`, `version`, `library`, `repository`, `remote_url`, `remote_sha`, `built`. Unavailable fields are `NA` rather than omitted, so a reader can see that the information was absent rather than forgotten.

See Also

`capture_environment()`, `environment_diff()`.

Examples

```
deps <- capture_dependencies(c("stats", "utils"))
deps[, c("package", "version")]

# a package with no repository field records that fact
is.na(capture_dependencies("stats")$repository)
```

---

capture_environment *Capture the Analysis Environment for a Manifest*

Description

Records the facts a replicator needs to rebuild the session: R version, platform, operating system, a UTC timestamp, and the versions of the requested packages.

Usage

```
capture_environment(packages = loadedNamespaces())
```

Arguments

packages	Character vector of package names to record. Defaults to every currently loaded namespace.

Value

A list with `r_version`, `platform`, `os`, `captured_utc`, and `packages` (a named character vector of versions).

Examples

```
# Record specific packages' versions alongside the session facts.
env <- capture_environment(c("stats", "utils"))
env$r_version
env$os
env$packages          # named character vector of versions

# Default captures every currently loaded namespace.
names(capture_environment())[1:4]
```

`cert_chain_verify` *Verify a certificate chain*

Description

Builds the path from a leaf certificate to a trust anchor you supply, verifying each signature, each validity window, and the constraints that stop a certificate being used for something it was not issued for.

Usage

```
cert_chain_verify(
  leaf,
  trust,
  intermediates = list(),
  at_time = Sys.time(),
  purpose = NULL,
  crls = list(),
  policies = NULL,
  revocation = c("supplied", "fetch", "none")
)
```

Arguments

`leaf`	The certificate to validate, from `cert_parse()` or anything `cert_parse()` accepts.
`trust`	A list of trust anchors – the certificates you have decided to believe. A chain that does not reach one of these fails.
`intermediates`	Optional further certificates to build the path through. They are not trusted by being supplied; they still have to verify.
`at_time`	The time to check validity windows at. Defaults to now.

purpose	Optional extended key usage the leaf must carry, as a name ("timeStamping", "codeSigning", ...) or an OID.
crls	Optional list of CRLs, as raw or paths, to check the chain against.
policies	Acceptable certificate policy OIDs. Supplied, the path must yield at least one of them after policy mapping and the constraints in RFC 5280 section 6.1; omitted, policies are processed but only reported as a failure when a certificate in the path requires an explicit policy.
revocation	"supplied" (the default) checks only the CRLs given in crls. "fetch" additionally retrieves CRLs from the distribution points in the certificates and queries OCSP responders – which makes verification depend on the network and discloses to the responder which certificates are being checked, so it is never the default. "none" skips revocation entirely.

Details

at_time is the point the validity windows are checked against, and it matters which one you pass. For a timestamp token the right answer is the time the token asserts: a token signed in 2020 by a certificate that expired in 2021 was validly signed, and checking it against today would reject it for no good reason. For a signature you are being asked to rely on now, pass now.

Value

A list of class bricklayer_certpath_check: ok, the path that was built, and a checks data frame.

See Also

cert_parse(), timestamp_verify().

Examples

```
## Not run:
res <- cert_chain_verify("tsa.crt", trust = "ca.crt",
                         purpose = "timeStamping")
res$ok
res$checks

## End(Not run)
```

---

cert_parse *Parse an X.509 certificate*

Description

Reads the fields a verifier needs: who issued it, who it is for, when it is valid, what key it carries, and what that key is allowed to do.

Usage

```
cert_parse(certificate)
```

Arguments

certificate	DER or PEM, as a raw vector or a path.

Value

A list of class bricklayer_certificate.

See Also

cert_chain_verify(), timestamp_verify().

Examples

```
# Certificates come from outside the package, so there is nothing to
# parse offline; this is the shape of the call.
## Not run:
cert <- cert_parse("tsa.crt")
cert$subject
cert$not_after
cert$extended_key_usage

## End(Not run)
```

---

chunk_file *Split a file into fixed-size chunks*

Description

Reads path as bytes and returns them as chunk strings suitable for merkle_root() and friends. The default 1 MiB chunk is a compromise: smaller chunks localise a change more precisely but make the tree and its proofs larger.

Usage

```
chunk_file(path, chunk_bytes = 1048576L)
```

Arguments

path	Path to an existing file.
chunk_bytes	Chunk size in bytes (default 1048576, i.e. 1 MiB).

Value

A list of raw vectors, in file order. A zero-length file gives an empty list. Chunks are returned as bytes rather than strings because a file is bytes: an R string cannot hold a zero byte, so a character chunk could not represent an arbitrary binary file at all.

See Also

`merkle_root()`, `sha512_file()`

Examples

```
p <- tempfile()
writeLines(rep("some capsule content", 50), p)

# One small chunk size to show the splitting.
ch <- chunk_file(p, chunk_bytes = 128)
length(ch)

# The chunks reconstruct the file and pin it as a Merkle root.
merkle_root(ch)

# They are the file's bytes, so they concatenate back to it exactly.
identical(unlist(ch), readBin(p, "raw", file.size(p)))

# Editing the file changes exactly one leaf.
unlink(p)
```

---

cite_capsule *Generate a Data Citation From Provenance*

Description

Builds a ready-to-paste data citation (plain text and BibTeX `@misc`) from a provenance object's `dataset` and `resource` blocks, using publisher, resource name, source system, retrieval date, license, the pinned URL, and a DOI when one is recorded (`dataset$doi`) .

Usage

```
cite_capsule(provenance)
```

Arguments

`provenance`	A provenance list as returned by `load_provenance()`.

Value

A list with `text` and `bibtex` character scalars, or `NULL` if `provenance` is `NULL`.

Examples

```
prov <- list(
  captured_at_utc = "2026-06-23T04:41:40Z",
  dataset = list(publisher = "Ontario Ministry of the Solicitor General",
                 licence_short = "OGL-Ontario",
                 package_slug = "data-on-inmates-in-ontario"),
  resource = list(name = "Restrictive Confinement - Detailed Dataset",
                  direct_url = "https://data.ontario.ca/example.csv")
)
cit <- cite_capsule(prov)

# Plain-text citation ready to paste.
cat(cit$text)

# BibTeX @misc entry for LaTeX bibliographies.
cat(cit$bibtex)

# NULL provenance returns NULL (composes safely).
cite_capsule(NULL)
```

---

clean_column_names *Tidy the column names of an ingested table*

Description

Converts names to a consistent, syntactically valid form: transliterated to ASCII, non-alphanumerics collapsed to a single separator, and duplicates disambiguated with a numeric suffix. The counterpart of `janitor::clean_names()`.

Usage

```
clean_column_names(
  data,
  case = c("snake", "lower_camel", "upper_camel", "screaming_snake", "none"),
  sep = "_"
)
```

Arguments

`data`	A data frame, or a character vector of names.
`case`	`"snake"` (default), `"lower_camel"`, `"upper_camel"`, `"screaming_snake"`, or `"none"` to normalise separators only.
`sep`	Separator for `"snake"` and `"screaming_snake"` (default `"_"`) .

Details

Open-data extracts arrive with names like "Total Population (2021)" and "% change", which need backticks everywhere and break silently when a re-release renames "% change" to "% change". Normalising once at ingestion makes the schema stable against that.

Record the mapping in the capsule manifest. Cleaning names changes what a downstream script must refer to, so an unrecorded cleaning is itself a reproducibility hazard – the returned object carries the original names in its "original_names" attribute for exactly that.

Value

The data frame with new names (and an "original_names" attribute), or the cleaned character vector.

Examples

```
clean_column_names(c("Total  Population (2021)", "% change",
                     "Ville / City", "dup", "dup"))

# Applied to a data frame, with the original names retained.
df <- data.frame(`Total Pop` = 1:2, `% change` = 3:4,
                 check.names = FALSE)
cleaned <- clean_column_names(df)
names(cleaned)
attr(cleaned, "original_names")

# Other cases.
clean_column_names(c("Total Pop"), case = "lower_camel")
clean_column_names(c("Total Pop"), case = "upper_camel")
clean_column_names(c("Total Pop"), case = "screaming_snake")
```

---

concentration *Concentration of a total across units*

Description

Concentration of a total across units

Usage

```
gini(x, na.rm = TRUE)

lorenz(x, na.rm = TRUE)

top_share(x, fractions = c(0.01, 0.05, 0.1, 0.25), na.rm = TRUE)
```

Arguments

x	Non-negative values, one per unit.
na.rm	Whether to drop missing values. They are dropped either way; this argument exists so the call reads the same as base R's.
fractions	Fractions of the units, largest first, to report the share of.

Details

Gini is the mean absolute difference between pairs of units over twice the mean, which is also twice the area between the Lorenz curve and the diagonal. Zero is a perfectly even spread; the maximum for n units is 1 - 1/n, not 1, so a Gini near 1 requires many units as well as an uneven spread.

Value

gini() a single number in [0, 1], NA when the total is zero. lorenz() a data frame of cumulative population and value shares, including the origin. top_share() a data frame of the requested fractions, the share each holds, and how many units that was.

References

Hedderich, J. and Sachs, L. (2020). *Applied Statistics: Methods Using R*. Springer-Verlag, Berlin Heidelberg. Section 3.14, p. 117, gives the construction used here: the units are placed at equal intervals on the horizontal axis and the cumulated, ascendingly ordered shares of the total on the vertical one, so that the curve is the diagonal exactly when p percent of the units account for p percent of the total, and sags further the greater the concentration.

See Also

hill_tail_index(), band_sensitivity()

Examples

```
# Ten units holding one each: no concentration.
gini(rep(1, 10))

# One unit holding everything: the maximum for ten units.
gini(c(rep(0, 9), 1))
1 - 1 / 10

placements <- c(rep(1, 1200), rep(3, 430), rep(8, 110), rep(20, 38))
gini(placements)

# What the most frequent few account for.
top_share(placements, c(0.01, 0.05, 0.1))

head(lorenz(placements))
```

core_blake2b *BLAKE2b digest, optionally keyed*

Description

BLAKE2b (RFC 7693): a modern cryptographic hash, faster than SHA-256 in software, that takes a key natively and produces any digest length from 1 to 64 bytes.

Usage

```
core_blake2b(x, key = NULL, length = 32L)
```

Arguments

x	A character vector or a raw vector.
key	Optional key, at most 64 bytes. NULL (default) gives the plain digest.
length	Digest length in bytes, 1 to 64 (default 32, matching SHA-256's width).

Details

The native key is the interesting part. A keyed BLAKE2b IS a message authentication code, with no HMAC wrapper and so no doubled hashing – `core_blake2b(msg, key = k)` does the job of `core_hmac_sha256()` at lower cost. Use SHA-256 or HMAC where interoperability with other tools matters; use this where it does not and speed does.

Value

A character vector of lowercase hex digests – one per element for character input, length-1 for raw input.

References

Saarinen MJ, Aumasson JP (2015). The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693. doi:10.17487/RFC7693

See Also

`core_sha256()`, `core_sha512()`, `core_hmac_sha256()`

Examples

```
# The RFC 7693 test vector for BLAKE2b-512 of "abc".
core_blake2b("abc", length = 64)

# 32 bytes by default, the same width as SHA-256.
core_blake2b("abc")
nchar(core_blake2b("abc"))
```

```
# Any digest length, which SHA-2 cannot do.
core_blake2b("abc", length = 8)

# Keyed, so it authenticates without an HMAC construction.
core_blake2b("manifest", key = "secret")
core_blake2b("manifest", key = "secret") ==
  core_blake2b("manifest", key = "other")

# Vectorised, and raw input hashes the same bytes.
core_blake2b(c("a", "b"))
identical(core_blake2b("abc"), core_blake2b(charToRaw("abc")))
```

`core_bootstrap_mean` *Bootstrap replicate means (C backend)*

Description

`B` resamples of `x`, drawn with replacement and each the same length as `x`, with the mean of every resample returned. The resampling uses the core's own 64-bit Mersenne Twister seeded by `seed`, NOT R's RNG, so a given `seed` reproduces the same replicates in every binding of the core and R's own random stream is left untouched.

Usage

```
core_bootstrap_mean(x, B = 1000L, seed = 42L)
```

Arguments

`x`	Numeric vector to resample.
`B`	Number of bootstrap replicates (default 1000).
`seed`	Seed for the core's generator (default 42).

Value

A numeric vector of length `B`: the replicate means.

Examples

```
set.seed(1)
x <- stats::rnorm(50, mean = 5)

reps <- core_bootstrap_mean(x, B = 500, seed = 7)
length(reps)

# The replicates centre on the sample mean, and their spread estimates
# the standard error.
c(sample = mean(x), bootstrap = mean(reps))
c(bootstrap_se = stats::sd(reps), formula_se = stats::sd(x) / sqrt(length(x)))
```

```
# A percentile confidence interval for the mean.
stats::quantile(reps, c(0.025, 0.975))

# Reproducible: the same seed gives the same replicates, and R's own
# random stream is not consumed.
identical(core_bootstrap_mean(x, 100, seed = 1),
          core_bootstrap_mean(x, 100, seed = 1))
```

`core_cov` *Covariance matrix of a numeric matrix (C backend)*

Description

Column covariances with the `n - 1` denominator, matching `stats::cov()`. Column names are carried through to both dimensions of the result.

Usage

```
core_cov(x)
```

Arguments

`x`	A numeric matrix or data frame of numeric columns (rows = observations, columns = variables).

Value

A symmetric `ncol(x)` by `ncol(x)` numeric matrix.

Examples

```
X <- cbind(a = c(1, 2, 3, 4), b = c(2, 4, 7, 8), c = c(5, 3, 2, 1))
core_cov(X)

# Agrees with stats::cov().
all.equal(core_cov(X), stats::cov(X))

# The diagonal is the column variances.
all.equal(diag(core_cov(X)), apply(X, 2, stats::var))

# Data frames are accepted.
core_cov(data.frame(u = 1:5, v = c(2, 1, 4, 3, 6)))
```

core_crc32 *CRC-32 checksum (C backend)*

Description

The CRC-32 of ITU V.42 and zip (reflected polynomial `0xEDB88320`) .

Usage

```
core_crc32(x)
```

Arguments

x	A character vector or a raw vector.

Details

A CRC is NOT a cryptographic digest: it detects accidental corruption – a truncated download, a flipped bit on disk – but anyone can construct a different input with the same value, so it must never be used where `core_sha256()` is meant. It is here because it is far cheaper than SHA-256 over gigabyte-scale capsule members, which makes it the right first pass when the question is only "did this file arrive intact".

Value

A numeric vector of unsigned 32-bit checksums (one per element for character input; length-1 for raw input). Returned as `double` rather than `integer` because values above `2^31 - 1` are not representable as an R integer.

See Also

`core_sha256()` for integrity against tampering rather than accident.

Examples

```
# The standard check value: CRC-32("123456789") is 0xCBF43926.
core_crc32("123456789")
core_crc32("123456789") == 0xCBF43926

# The empty input has checksum zero.
core_crc32("")

# Vectorised, and sensitive to a single changed byte.
core_crc32(c("brick", "brack"))

# Raw bytes work the same way.
core_crc32(charToRaw("123456789"))
```

`core_gamma_cdf` *Regularized incomplete gamma function (C backend)*

Description

The lower regularized incomplete gamma function $P(a, x)$, which is the CDF of a Gamma distribution with shape `a` and unit rate. Exposed because it is the building block of the chi-square tail used by `drift_chisq()` and `benford_test()`.

Usage

```
core_gamma_cdf(shape, x)
```

Arguments

`shape`	Shape parameter a (length-1, > 0).
`x`	Numeric vector of quantiles (>= 0).

Value

A numeric vector the length of `x`, each entry in \ [0, 1].

[0, 1]: R:0,%201%5C

Examples

```
core_gamma_cdf(3.5, c(0.5, 1, 4, 12))

# Identical to the unit-rate gamma CDF.
all.equal(core_gamma_cdf(3.5, c(0.5, 1, 4, 12)),
          stats::pgamma(c(0.5, 1, 4, 12), shape = 3.5))

# Shape 1 is the exponential distribution.
all.equal(core_gamma_cdf(1, c(0.5, 2)), stats::pexp(c(0.5, 2)))

# A chi-square tail on k degrees of freedom is 1 - P(k/2, q/2).
1 - core_gamma_cdf(2 / 2, 5.99 / 2)      # about 0.05 on 2 df
```

`core_hawkes_nll` *Hawkes-process negative log-likelihood (C backend)*

Description

The negative log-likelihood of a univariate self-exciting Hawkes process with constant baseline on `[0, horizon]`, for the event times `times`. A Hawkes process is the natural model for arrivals that trigger further arrivals – repeat calls to a service, aftershocks, retweet cascades, revisions to an open-data release.

Usage

```
core_hawkes_nll(
  times,
  horizon,
  kernel = c("exponential", "weibull", "lomax", "gamma"),
  par
)
```

Arguments

`times`	Sorted numeric vector of event times in `[0, horizon]`.
`horizon`	End of the observation window (length-1, > 0).
`kernel`	One of `"exponential"`, `"weibull"`, `"lomax"`, `"gamma"`.
`par`	Numeric parameter vector, as described above: length 3 for `"exponential"`, length 4 for the others.

Details

Four triggering kernels are available. `"exponential"` is memoryless and evaluates by an O(n) recursion; the other three are not, so they cost O(n^2).

Parameters are passed on the scales the kernel is defined on: `par = c(a0, eta, ...)` where `a0` is the LOG baseline intensity ($\nu = e^{a0}$) and `eta` the branching ratio in (0, 1) – the expected number of children per event, so the process is stationary only for `eta < 1`. The remaining entries are the kernel's own shape parameters: `beta` (exponential), `alpha, lambda` (Weibull), `alpha, c` (Lomax), `alpha, beta` (gamma).

Value

A length-1 numeric: the negative log-likelihood, to be MINIMISED. A parameter set outside the core's feasible region returns the sentinel `1e12` rather than erroring, so the value can be handed straight to `stats::optim()` without the optimiser walking off the domain.

References

Hawkes AG (1971). Spectra of some self-exciting and mutually exciting point processes. *Biometrika* 58(1), 83–90. doi:10.1093/biomet/58.1.83

Examples

```
set.seed(4)
times <- sort(stats::runif(40, 0, 10))

# Exponential kernel: log-baseline -0.5, branching 0.3, decay 1.2.
core_hawkes_nll(times, 10, "exponential", c(-0.5, 0.3, 1.2))

# Lower is better, so this is what an optimiser minimises.
nll <- function(p) core_hawkes_nll(times, 10, "exponential", p)
fit <- stats::optim(c(-0.5, 0.3, 1.2), nll)
fit$par
```

```
# A branching ratio at or above 1 is not a stationary process, and is
# reported as the infeasible sentinel rather than a number.
core_hawkes_nll(times, 10, "exponential", c(-0.5, 1.5, 1.2)) == 1e12

# The heavier-tailed kernels take two shape parameters.
core_hawkes_nll(times, 10, "gamma", c(-0.5, 0.3, 2, 1.5))
```

`core_ipw_weights` *Trimmed inverse-probability weights (C backend)*

Description

The inverse-probability-of-treatment weights $1/e$ for the treated and $1/(1-e)$ for the untreated, with the propensity score clamped into `[trim_lo, trim_hi]` FIRST. Clamping matters: an untrimmed score near 0 or 1 produces a weight large enough for one observation to dominate the entire estimate.

Usage

```
core_ipw_weights(treat, propensity, trim_lo = 0.01, trim_hi = 0.99)
```

Arguments

`treat`	Numeric or logical treatment indicator; `1` / `TRUE` is treated.
`propensity`	Numeric vector of propensity scores, the same length as `treat`.
`trim_lo`, `trim_hi`	Clamp bounds for the score (defaults 0.01 and 0.99).

Value

A numeric vector of weights the length of `treat`.

Examples

```
treat <- c(1, 0, 1, 0)
e <- c(0.5, 0.25, 0.02, 0.9)

core_ipw_weights(treat, e)

# A balanced score gives weight 2 to either arm.
core_ipw_weights(c(1, 0), c(0.5, 0.5))

# Without trimming the third observation would carry weight 50; the
# default clamp holds it to 100 at the 0.01 floor, and a looser floor
# tames it further.
core_ipw_weights(treat, e, trim_lo = 0.10)

# Logical treatment indicators work too.
core_ipw_weights(c(TRUE, FALSE), c(0.4, 0.4))
```

`core_moments` *Mean, variance, skewness and kurtosis in one pass (C backend)*

Description

A single streaming pass (Welford's recurrence, extended to the third and fourth central moments) over `x`. One pass matters for capsule members large enough that reading the column twice is the expensive part.

Usage

```
core_moments(x)
```

Arguments

`x`	Numeric vector (coerced with `as.numeric()`) .

Details

The variance uses the `n - 1` denominator, matching `stats::var()`. The shape statistics use the *sample moment* definitions $m_3/m_2^{3/2}$ and $m_4/m_2^2 - 3$, with m_k the k-th central moment divided by `n` – so kurtosis is reported as EXCESS kurtosis and a normal sample sits near zero, not near three.

Value

A named length-4 numeric: `mean`, `variance`, `skewness`, `kurtosis`. `skewness` needs at least 3 observations and `kurtosis` at least 4; both are `NaN` below that, as is everything if `x` contains NA/NaN.

References

Welford BP (1962). Note on a method for calculating corrected sums of squares and products. *Technometrics* 4(3), 419–420. doi:10.1080/00401706.1962.10490022

Examples

```
core_moments(c(2, 4, 4, 4, 5, 5, 7, 9))

# The first two entries agree with base R.
m <- core_moments(1:10)
all.equal(m["mean"], mean(1:10))
all.equal(m["variance"], stats::var(1:10))

# A symmetric sample has no skew; excess kurtosis is near 0 for normal
# data and positive for a heavy-tailed sample.
core_moments(c(-2, -1, 0, 1, 2))["skewness"]
core_moments(c(rep(0, 20), -8, 8))["kurtosis"] > 0

# Too short to define a shape statistic: NaN rather than a guess.
core_moments(c(1, 2))
```

core_normal_logpdf *Normal log-density (C backend)*

Description

The logarithm of the normal density, computed directly rather than as `log(dnorm(x))`, so it stays finite far into the tails where the density itself underflows to zero.

Usage

```
core_normal_logpdf(x, mean = 0, sd = 1)
```

Arguments

`x`	Numeric vector of quantiles.
`mean`	Distribution mean (length-1, default 0).
`sd`	Distribution standard deviation (length-1, default 1, > 0).

Value

A numeric vector the length of `x`. Equivalent to `stats::dnorm(x, mean, sd, log = TRUE)`.

Examples

```
core_normal_logpdf(c(-1, 0, 1))

# Identical to stats::dnorm(log = TRUE).
all.equal(core_normal_logpdf(-2:2, 0.3, 1.7),
          stats::dnorm(-2:2, 0.3, 1.7, log = TRUE))

# Still finite where the density itself underflows to zero.
stats::dnorm(50)                  # 0
core_normal_logpdf(50)            # about -1251
```

core_normal_pdf *Normal density (C backend)*

Description

Vectorised over `x`; `mean` and `sd` are length-1.

Usage

```
core_normal_pdf(x, mean = 0, sd = 1)
```

Arguments

x	Numeric vector of quantiles.
mean	Distribution mean (length-1, default 0).
sd	Distribution standard deviation (length-1, default 1, > 0).

Value

A numeric vector the length of x. Equivalent to stats::dnorm(x, mean, sd).

Examples

```
# Standard normal density at a few quantiles.
core_normal_pdf(c(-1, 0, 1))

# Peak of the standard normal is 1/sqrt(2*pi) at x = 0.
core_normal_pdf(0)

# Shift and scale via `mean` and `sd`.
core_normal_pdf(5, mean = 5, sd = 2)       # peak of N(5, 2)
core_normal_pdf(c(0, 5, 10), mean = 5, sd = 2)

# Identical to stats::dnorm().
all.equal(core_normal_pdf(-2:2, 0, 1), stats::dnorm(-2:2, 0, 1))
```

---

core_sha256 *SHA-256 hex digest (C backend)*

Description

Hashes character or raw input with the self-contained SHA-256 in the rmoriebricklayer core. For a character vector each element is hashed as its UTF-8/native bytes; for a raw vector the raw bytes are hashed. This is the same routine sibling packages use for provenance via LinkingTo: rmoriebricklayer.

Usage

```
core_sha256(x)
```

Arguments

x	A character vector or a raw vector.

Value

A character vector of 64-character lowercase hex digests (one per element for character input; length-1 for raw input).

Examples

```
# Hash a character scalar (NIST test vector for "abc").
core_sha256("abc")
# ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

# Vectorised over character input: one digest per element.
core_sha256(c("abc", "def"))

# Raw input hashes the bytes directly; identical to the character form.
identical(core_sha256("abc"), core_sha256(charToRaw("abc")))

# Fingerprint an arbitrary object via its serialization.
core_sha256(serialize(list(a = 1L, b = "x"), NULL))
```

---

core_sha512 *SHA-512 hex digest (C backend)*

Description

Hashes character or raw input with the self-contained SHA-512 (FIPS 180-4) in the compiled core. Use it over `core_sha256()` when a pin has to outlive the capsule by decades: the wider digest leaves more margin, including against a quantum adversary, for whom Grover's algorithm halves the effective preimage exponent.

Usage

```
core_sha512(x)
```

Arguments

x	A character vector or a raw vector.

Value

A character vector of 128-character lowercase hex digests (one per element for character input; length-1 for raw input).

See Also

`core_sha256()`, `core_crc32()`, `sha512_file()`

Examples

```
# FIPS 180-4 test vector for "abc".
core_sha512("abc")

# Vectorised over character input.
core_sha512(c("abc", "def"))
```

```
# Raw input hashes the bytes directly, and agrees with the character
# form for the same bytes.
identical(core_sha512("abc"), core_sha512(charToRaw("abc")))

# Twice the digest width of SHA-256.
c(sha256 = nchar(core_sha256("abc")), sha512 = nchar(core_sha512("abc")))
```

`core_weighted` *Weighted mean and variance (C backend)*

Description

The weights are treated as RELIABILITY weights (how precisely each observation is known), so the variance carries the bias correction $\sum w - \sum w^2 / \sum w$ in the denominator rather than `n - 1`. With every weight equal to 1 it reduces exactly to `stats::var()`.

Usage

```
core_weighted(x, w)
```

Arguments

`x`	Numeric vector of observations.
`w`	Numeric vector of non-negative weights, the same length as `x`.

Value

A named length-2 numeric: `mean` and `variance`.

Examples

```
x <- c(10, 20, 30, 40)
w <- c(1, 1, 2, 4)
core_weighted(x, w)

# The mean agrees with base R.
all.equal(core_weighted(x, w)[["mean"]], stats::weighted.mean(x, w))

# Equal weights recover the unweighted variance.
all.equal(core_weighted(x, rep(1, 4))[["variance"]], stats::var(x))

# A single dominant weight pulls the mean onto that observation.
core_weighted(x, c(1, 1, 1, 1000))[["mean"]]
```

`correlation_table` *Full pairwise correlation table*

Description

Every numeric pair's correlation in long form – one row per pair, which is easier to sort, filter and join than a matrix. The counterpart of `corrr::correlate()`.

Usage

```
correlation_table(data, method = c("spearman", "pearson"), min_pairs = 3L)
```

Arguments

`data`	A data frame; non-numeric columns are ignored.
`method`	`"spearman"` (default) or `"pearson"`. See `top_correlations()` for why the rank correlation is the default.
`min_pairs`	Minimum complete pairs required before a correlation is computed (default 3). Below it the pair is `NA` rather than a number computed from almost nothing.

Value

A data frame of class `bricklayer_cortable` with `x`, `y`, `correlation` and `n_pairs`.

See Also

`top_correlations()` for just the strongest, `core_cov()` for the matrix form.

Examples

```
set.seed(1)
df <- data.frame(a = stats::rnorm(100), b = stats::rnorm(100))
df$c <- df$a + stats::rnorm(100, sd = 0.2)

correlation_table(df)

# n_pairs shows how much data each figure rests on.
gappy <- df
gappy$a[1:80] <- NA
correlation_table(gappy)

# A pair with too little overlap is NA, not a number from nothing.
thin <- df
thin$a[1:99] <- NA
correlation_table(thin)$correlation
```

`count_trend` *Trend in a count series, as a rate ratio per period*

Description

Fits a Poisson log-linear trend by iteratively reweighted least squares and reports the multiplicative change per period, which is what a count series' trend actually is. An offset carries the denominator when the exposure varies.

Usage

```
count_trend(y, x = NULL, offset = NULL, conf_level = 0.95)
```

Arguments

`y`	Counts, in period order.
`x`	Periods. Defaults to the position.
`offset`	Exposure for each period – a population, a number of admissions, a number of days. The trend is then in the rate rather than in the count.
`conf_level`	Confidence level for the rate ratio.

Details

The dispersion is reported because a Poisson fit assumes it is one. When it is well above one the interval is too narrow, and the quasi-Poisson interval – which scales the standard error by the square root of the dispersion – is returned instead, with `overdispersed` set.

Value

A list with `rate_ratio` (per period), its interval, `p_value`, the fitted values, the dispersion, and `overdispersed`.

References

Bilder, C. R. and Loughin, T. M. *Analysis of Categorical Data with R*, 2nd edn. Chapman and Hall/CRC, on the quasi-likelihood treatment of an overdispersed Poisson fit: the variance is scaled by an estimated dispersion, which widens the interval while leaving the point estimate alone. That is the behaviour reported here through `dispersion` and `overdispersed`.

Examples

```
# A count falling by about 15% a year.
set.seed(3)
y <- stats::rpois(8, lambda = 200 * 0.85^(0:7))
fit <- count_trend(y)
round(fit$rate_ratio, 3)

# With a varying denominator the trend is in the rate.
count_trend(c(20, 25, 30), offset = c(1000, 1500, 2500))$rate_ratio
```

`cramers_v` *Association in a contingency table*

Description

Cramer's V, with the bias correction of Bergsma (2013) available, and the small-expected-count condition reported rather than assumed away.

Usage

```
cramers_v(tbl, bias_correct = TRUE, min_expected = 5)
```

Arguments

`tbl`	A table or matrix of counts.
`bias_correct`	Whether to apply Bergsma's correction, which removes most of V's upward bias in a sparse table. Worth having: uncorrected V on a sparse table reports association that is an artefact of the table's size.
`min_expected`	Expected count below which the chi-square approximation is unreliable. Reported, not enforced.

Value

A list with `v`, `chisq`, `df`, `p_value`, `n`, `min_expected`, and `cells_below`, the number of cells whose expected count falls under the threshold. When any does, `p_value` comes from a Monte Carlo permutation instead of the chi-square approximation, and `method` says which was used.

References

Bergsma, W. (2013). A bias-correction for Cramer's V and Tschuprow's T. *Journal of the Korean Statistical Society* 42(3), 323-328. (Not in the local corpus; cited from the published paper.)

Examples

```
tbl <- rbind(c(120, 80), c(40, 160))
cramers_v(tbl)

# No association gives a V near zero.
cramers_v(rbind(c(100, 100), c(100, 100)))$v

# A sparse table's uncorrected V overstates the association.
sparse <- rbind(c(3, 1), c(1, 3))
c(raw = cramers_v(sparse, bias_correct = FALSE)$v,
  corrected = cramers_v(sparse)$v)
```

`derive_key` *Derive a key from a passphrase*

Description

PBKDF2-HMAC-SHA256 (RFC 8018): stretches a passphrase into a key of full width by iterating a keyed hash, so guessing the passphrase costs `iterations` times more than a single hash would.

Usage

```
derive_key(passphrase, salt, iterations = 100000L, length = 32L)
```

Arguments

`passphrase`	Passphrase, as a length-1 character or raw vector.
`salt`	Unique, non-secret salt (length-1 character or raw). Use `random_bytes()` to make one, and store it beside the key.
`iterations`	Iteration count (default 100000, minimum 1).
`length`	Derived key length in bytes (default 32).

Details

The `salt` must be UNIQUE per key and need not be secret. Its job is to make precomputation useless: without one, a single table of common passphrases attacks every key at once.

`iterations` is the cost knob. The default 100,000 is a reasonable 2020s floor for an interactive use; raise it for anything valuable, and record the value you used, since verification must repeat it exactly.

PBKDF2 resists brute force by ITERATION only, not by memory. Where a memory-hard function is available (`argon2` in **sodium**, `bcrypt_pbkdf` in **openssl**) prefer it for passwords a human chose. PBKDF2 is here because it needs nothing beyond the bundled SHA-256, so it works wherever this package works.

Value

A length-1 character vector: the key as lowercase hex.

References

Moriarty K, Kaliski B, Rusch A (2017). PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018. doi:10.17487/RFC8018

See Also

`random_bytes()`, `core_hmac_sha256()`, `capsule_sign()`

Examples

```
# The published PBKDF2-HMAC-SHA256 vector: "password", "salt", 1 round.
derive_key("password", "salt", iterations = 1)

# Deterministic, so verification can repeat it.
identical(derive_key("pw", "s", 1000), derive_key("pw", "s", 1000))

# The salt, the passphrase and the iteration count all change the key.
derive_key("pw", "salt-a", 1000) == derive_key("pw", "salt-b", 1000)
derive_key("pw", "s", 1000) == derive_key("pw", "s", 2000)

# Use it to sign a manifest from a passphrase rather than raw bytes.
salt <- paste(format(random_bytes(16)), collapse = "")
key <- derive_key("correct horse battery staple", salt)
sig <- capsule_sign("manifest-digest", key, scheme = "hmac")
capsule_verify("manifest-digest", sig, key)

# A longer key is a prefix-consistent extension of a shorter one.
identical(substring(derive_key("pw", "s", 10, length = 64), 1, 64),
          derive_key("pw", "s", 10, length = 32))
```

`digest_object` *Digest an arbitrary R object*

Description

Fingerprints any R object by hashing its serialization, so a list, a data frame, a function or a fitted model all get a stable digest. The counterpart of `digest::digest()`, computed with this package's own hashes.

Usage

```
digest_object(x, algo = c("sha256", "sha512", "blake2b", "crc32"), key = NULL)
```

Arguments

`x`	Any R object.
`algo`	`"sha256"` (default), `"sha512"`, `"blake2b"` or `"crc32"`.
`key`	Optional key, for `"blake2b"` only: gives a keyed fingerprint that only a key holder can reproduce.

Details

Serialization is pinned to version 2 with XDR byte order, so the digest is the same on a big-endian machine as on a little-endian one. Two objects that are `identical()` produce the same digest; two that merely print the same need not, because attributes are part of the serialization.

Value

A length-1 character vector (or numeric for `"crc32"`) .

See Also

`core_sha256()` for hashing text or bytes directly, `bricklayer_json_serialize()` for a readable lossless form.

Examples

```
digest_object(list(a = 1L, b = "x"))

# Stable across calls, and sensitive to any change.
identical(digest_object(1:10), digest_object(1:10))
digest_object(1:10) == digest_object(1:11)

# Attributes are part of the object, so they are part of the digest.
digest_object(matrix(1:6, nrow = 2)) == digest_object(1:6)

# Any of the hashes, and a keyed fingerprint.
digest_object(mtcars, algo = "sha512")
digest_object(mtcars, algo = "crc32")
digest_object(mtcars, algo = "blake2b", key = "secret")

# A data frame's digest pins the data, so it can go in a manifest.
digest_object(data.frame(x = 1:3))
```

---

download_data *Download a File*

Description

Thin wrapper around `utils::download.file()` that returns the target path invisibly so it composes in pipelines.

Usage

```
download_data(url, target_path, mode = "wb", quiet = FALSE)
```

Arguments

`url`	URL to download.
`target_path`	Destination path on disk.
`mode`	Write mode passed to `utils::download.file()`; defaults to `"wb"` (binary) for cross-platform safety.
`quiet`	Logical; suppress progress output. Defaults to `FALSE`.

Value

The `target_path`, returned invisibly.

Examples

```
# try(): a live download must fail gracefully on an offline check machine.
dest <- try(download_data("https://cloud.r-project.org/",
                          tempfile(fileext = ".html"), quiet = TRUE))
if (!inherits(dest, "try-error")) file.exists(dest)
```

`drift_chisq` *Chi-square test for a categorical column*

Description

Pearson's chi-square goodness-of-fit statistic comparing observed category counts with the proportions a capsule was pinned against. Use it where `drift_ks()` cannot apply, because the column is a factor or a set of codes rather than a number.

Usage

```
drift_chisq(observed, expected)
```

Arguments

`observed`	Named numeric vector of counts in the new sample, or a factor/character vector to be tabulated.
`expected`	Named numeric vector of reference counts or proportions, or a factor/character vector to be tabulated. Rescaled to the total of `observed`.

Details

Categories present in one argument and not the other are aligned by name, so a vanished category registers as a shortfall against its expected count.

A category that OCCURS but which the reference gives probability zero contradicts the pinned distribution outright, and its chi-square term is unbounded; `statistic` is then `Inf` and `p_value` is `0`. If a new category is a legitimate possibility rather than a contradiction – which it usually is when the reference is itself a finite sample – use `drift_homogeneity()` instead.

Value

A named length-3 numeric: `statistic`, `df`, `p_value`.

See Also

`stats::chisq.test()` for the full test object.

Examples

```
ref <- c(a = 50, b = 30, c = 20)

# Counts matching the reference proportions: nothing to report.
drift_chisq(c(a = 100, b = 60, c = 40), ref)

# A reallocated mix is detected.
drift_chisq(c(a = 40, b = 60, c = 100), ref)

# Agrees with stats::chisq.test().
o <- c(a = 40, b = 60, c = 100)
all.equal(drift_chisq(o, ref)[["statistic"]],
          as.numeric(stats::chisq.test(o, p = ref / sum(ref))$statistic))

# Raw vectors are tabulated for you.
drift_chisq(c("a", "a", "b", "b"), c("a", "b"))

# A category the reference rules out, but which occurs, is a flat
# contradiction rather than a large finite statistic.
drift_chisq(c("a", "a", "b", "d"), c("a", "a", "b", "b"))

# When a new category is legitimate, compare two samples instead.
drift_homogeneity(c("a", "a", "b", "b"), c("a", "a", "b", "d"))
```

---

drift_homogeneity *Chi-square test of homogeneity for two categorical samples*

Description

Tests whether two SAMPLES were drawn from the same categorical distribution, by Pearson's chi-square on the 2-by-k contingency table of their counts.

Usage

```
drift_homogeneity(x, y)
```

Arguments

x, y	Factor or character vectors (or named count vectors) – the reference and the new sample.

Value

A named length-3 numeric: `statistic`, `df`, `p_value`.

Why this and not drift_chisq()

drift_chisq() compares observed counts against a distribution taken as KNOWN – proportions fixed by a specification. When the reference is itself a finite sample, that treatment ignores the reference's own sampling error, understates the variance of the comparison, and so reports drift too readily. A homogeneity test estimates the shared distribution from the pooled margins and carries the uncertainty of both samples, which is the right test when comparing a pinned extract with a fresh fetch. capsule_drift() therefore uses this one.

Categories present in only one sample are aligned by name and given zero counts, so an appearing or vanishing level registers.

See Also

drift_chisq() for a known reference distribution, stats::chisq.test() for the full test object.

Examples

```
set.seed(1)
a <- sample(c("x", "y", "z"), 300, TRUE)
b <- sample(c("x", "y", "z"), 300, TRUE)

# Two draws from the same distribution: no evidence of a difference.
drift_homogeneity(a, b)

# A reallocated mix is detected.
drift_homogeneity(a, sample(c("x", "y", "z"), 300, TRUE,
                            prob = c(0.7, 0.2, 0.1)))

# Agrees with stats::chisq.test() on the 2-by-k table.
tab <- rbind(table(a), table(b))
all.equal(drift_homogeneity(a, b)[["statistic"]],
          as.numeric(stats::chisq.test(tab)$statistic))

# It is more conservative than treating the reference as known, which
# is exactly the point.
drift_homogeneity(a, b)[["p_value"]] >= drift_chisq(b, a)[["p_value"]]
```

---

drift_ks *Two-sample Kolmogorov-Smirnov test (C backend)*

Description

The largest vertical gap between the two empirical distribution functions, with the asymptotic two-sided p-value from the Kolmogorov distribution. Distribution-free: it assumes nothing about the shape of either sample, which is what makes it the right first test on a column whose distribution was never specified.

Usage

```
drift_ks(x, y)
```

Arguments

x, y	Numeric vectors, the reference and the new sample.

Details

The p-value is the ASYMPTOTIC one, $2\sum_k (-1)^{k-1} e^{-2k^2 t^2}$ with $t = \sqrt{n_{\text{eff}}} D$. It is accurate for moderate samples and conservative for small ones; for an exact small-sample p-value use `stats::ks.test()`. Ties are handled by comparing the two EDFs at each distinct value, so tied data does not produce a warning the way `ks.test()` does.

Value

A named length-3 numeric: `statistic` (the KS D) , `p_value`, and `n_eff` (the harmonic-style effective size $n_x n_y/(n_x + n_y)$).

See Also

`drift_psi()`, `drift_chisq()`, `capsule_drift()`

Examples

```
set.seed(1)
ref <- stats::rnorm(200)

# The same distribution: a small D and a large p-value.
drift_ks(ref, stats::rnorm(200))

# A shifted distribution is detected.
drift_ks(ref, stats::rnorm(200, mean = 0.8))

# So is a change in spread alone, which a mean comparison would miss.
drift_ks(ref, stats::rnorm(200, sd = 2.5))

# The statistic agrees with stats::ks.test().
a <- stats::rnorm(60); b <- stats::rnorm(45, 0.6)
all.equal(drift_ks(a, b)[["statistic"]],
          as.numeric(suppressWarnings(stats::ks.test(a, b))$statistic))

# Identical samples have nothing to report.
drift_ks(ref, ref)[["statistic"]]
```

---

drift_psi *Population stability index and Jensen-Shannon divergence (C backend)*

Description

Two summaries of how far a new binned distribution has moved from a reference one.

Usage

```
drift_psi(x, y, bins = 10L, eps = 1e-06)
```

Arguments

x, y	Numeric vectors, the reference and the new sample. Binned on the quantiles of x, so the reference defines the bins.
bins	Number of bins (default 10).
eps	Floor applied to empty bins in the PSI (default 1e-6).

Details

The population stability index is $\sum_i (p_i - q_i)\log(p_i/q_i)$ – the symmetrised Kullback-Leibler divergence of the two discrete distributions. The conventional reading, from credit-risk monitoring where it originates, is that below 0.1 is stable, 0.1 to 0.25 warrants a look, and above 0.25 is a material shift.

The Jensen-Shannon divergence is $\frac{1}{2}KL(p\|m)+\frac{1}{2}KL(q\|m)$ with m the mixture $(p+q)/2$. Unlike PSI it is bounded – by $\log 2$ in nats – so it is comparable across columns with different numbers of bins, and it is finite even when a category is absent from one side.

Empty bins are floored at eps for the PSI only, since $\log(0)$ would otherwise send it to infinity on a single missing category.

Value

A named length-2 numeric: psi and js_divergence.

References

Wu D, Olson DL (2010). Enterprise risk management: coping with model risk in a large bank. *Journal of the Operational Research Society* 61(2), 179–190. doi:10.1057/jors.2008.144

Lin J (1991). Divergence measures based on the Shannon entropy. *IEEE Transactions on Information Theory* 37(1), 145–151. doi:10.1109/18.61115

Examples

```
set.seed(2)
ref <- stats::rnorm(500)

# Same distribution: both indices near zero.
drift_psi(ref, stats::rnorm(500))

# A shift both indices register.
drift_psi(ref, stats::rnorm(500, mean = 1))

# A sample compared with itself has moved nowhere at all.
drift_psi(ref, ref)

# The Jensen-Shannon divergence is bounded by log(2), whatever the
# shift, which is what makes it comparable across columns.
```

```
drift_psi(c(1, 1, 1), c(9, 9, 9))[["js_divergence"]] <= log(2)
```

`duplicate_rows` *Duplicated rows, with their groups*

Description

Returns the rows that share a combination of `columns` with at least one other row, grouped so the duplicates sit together. The counterpart of `janitor::get_dupes()`.

Usage

```
duplicate_rows(data, columns = NULL)
```

Arguments

`data`	A data frame.
`columns`	Columns defining a duplicate (default: all of them).

Details

`rule_distinct_rows()` tells you THAT there are duplicates, which is what a validation gate needs. This shows you WHICH, which is what fixing them needs – and a duplicate is usually a join that fanned out or a re-release appended rather than replaced, both of which are visible only once the offending rows are in front of you.

Value

The duplicated rows, ordered by group, with a `dupe_count` column giving each group's size. Zero rows when there are none.

See Also

`rule_distinct_rows()`, `rule_unique()`

Examples

```
df <- data.frame(id = c(1, 2, 2, 3, 3, 3),
                 value = c("a", "b", "b", "c", "d", "c"),
                 stringsAsFactors = FALSE)

# Duplicated on every column.
duplicate_rows(df)

# Duplicated on the identifier alone, which catches more.
duplicate_rows(df, "id")

# No duplicates gives zero rows, not an error.
duplicate_rows(data.frame(x = 1:3))
```

`eb_rates` *Empirical Bayes rates, shrunk toward the overall experience*

Description

A small area's rate is mostly noise, so ranking areas by their raw rates puts the smallest areas at both ends of the table by construction. This borrows strength across areas: each rate is pulled toward the overall one by an amount that depends on how little information the area carries.

Usage

```
eb_rates(observed, expected, area = NULL)
```

Arguments

`observed`	Observed counts.
`expected`	Expected counts.
`area`	Optional labels.

Details

The Clayton-Kaldor construction: the area-specific relative risks are taken to come from a gamma prior, whose two parameters are estimated from the observed and expected counts by the method of moments, and the posterior mean `(O + nu) / (E + alpha)` is reported. Where the expected count is large the data dominate and the estimate barely moves; where it is small the prior does, which is the intended behaviour and not a defect.

When the between-area variance estimate comes out at or below zero there is no evidence of any real variation between areas, and every estimate collapses to the overall rate. That is reported through `shrinkage` rather than hidden.

Value

A data frame with the raw `sir`, the shrunk `eb`, the `shrinkage` applied (zero means untouched, one means replaced by the overall rate), and the fitted prior's `nu` and `alpha`.

References

Clayton, D. and Kaldor, J. (1987). Empirical Bayes estimates of age-standardized relative risks for use in disease mapping. *Biometrics* 43(3), 671-681.

Lawson, A. B. *Using R for Bayesian Spatial and Spatio-Temporal Health Modeling*. Chapman and Hall/CRC, which cites Clayton and Kaldor as the empirical-Bayes approximation in the development of Bayesian disease mapping.

See Also

`sir()`, `funnel_limits()`

Examples

```
# Three areas, one of them tiny. The tiny area's raw ratio is
# extreme; its shrunk one is not.
eb_rates(observed = c(30, 45, 2), expected = c(25, 50, 0.5),
         area = c("North", "South", "Tiny"))
```

`environment_diff` *Compare two captured environments*

Description

Reports what changed between the environment captured by one run and another: the R version, the platform, and every package whose version differs, was added, or disappeared.

Usage

```
environment_diff(a, b)
```

Arguments

a, b	Environment records from `capture_environment()`, or whole manifests containing an `environment` element.

Details

This is the question a failed reproduction actually raises. A manifest records the environment; comparing two manifests by eye across a few hundred packages does not scale, and the one line that matters – a dependency that moved a minor version – is exactly what gets missed.

Value

A list of class `bricklayer_env_diff`: `identical` (logical), `r_version` (a length-2 character vector when they differ, else `NULL`) , `platform` (likewise), and `packages` (a data frame of `package`, `a`, `b`, `change`, where `change` is `"added"`, `"removed"` or `"changed"`) .

See Also

`capture_environment()`, `make_manifest()`

Examples

```
a <- capture_environment()
b <- a

# An environment matches itself.
environment_diff(a, b)$identical

# Stage a moved dependency and a removed one.
```

```
if (length(b$packages)) {
  nm <- names(b$packages)[1]
  b$packages[[nm]] <- "0.0.0"
  environment_diff(a, b)$packages
}

# Whole manifests are accepted, not just the environment block.
m <- make_manifest(list(run = "demo"))
environment_diff(m, m)$identical
```

`evalue_rr` *How strong would an unmeasured confounder have to be*

Description

The E-value of VanderWeele and Ding (2017): the minimum strength of association, on the risk-ratio scale, that an unmeasured confounder would need with BOTH the exposure and the outcome to explain away an observed risk ratio.

Usage

```
evalue_rr(rr, lo = NULL, hi = NULL, true = 1)
```

Arguments

`rr`	Observed risk ratio, greater than 0. Protective effects (below 1) are inverted first, as the measure is symmetric.
`lo`, `hi`	Optional confidence limits on the same scale. The limit nearer the null is the one used.
`true`	The value to move the estimate to. Defaults to 1, the null.

Details

It answers the question a covariate list cannot: not whether the analysis adjusted for the right things, but how much unmeasured confounding it would take to move the result to nothing. An E-value of 1.2 says very little would be needed; an E-value of 5 says a confounder five times more common in the exposed group AND five times more associated with the outcome would have to have gone unnoticed.

The E-value for the confidence limit is the one to report alongside it: a large point-estimate E-value with a limit E-value of 1 means the interval already includes no effect, and no confounding is needed at all.

Value

A named numeric vector: `evalue_point` and, when limits are given, `evalue_limit`.

References

VanderWeele, T. J., and Ding, P. (2017). Sensitivity Analysis in Observational Research: Introducing the E-Value. *Annals of Internal Medicine* 167(4), 268-274. doi:10.7326/M162607

See Also

`capsule_falsify()` for controls that use the data, where this uses none.

Examples

```
# a risk ratio of 2 needs a confounder associated by 3.41 with both
evalue_rr(2)

# a protective effect is inverted, so 0.5 gives the same answer
evalue_rr(0.5)

# an interval that already includes the null needs nothing
evalue_rr(2, lo = 0.9, hi = 4.4)

# and a strong result with a limit well above the null is harder to
# explain away
evalue_rr(3, lo = 2.1, hi = 4.3)
```

---

expand_bands *Expand a banded frequency table into per-unit values*

Description

Expand a banded frequency table into per-unit values

Usage

```
expand_bands(bands, counts, ...)
```

Arguments

bands	A data frame from `parse_bands()`, or labels to parse.
counts	How many units fall in each band.
...	Passed to `band_values()`.

Value

A numeric vector with one entry per unit.

Examples

```
x <- expand_bands(c("1", "2 to 5", "Greater than 5"),
                  counts = c(10, 4, 2), open_upper_cap = 12)
table(x)

gini(x)
```

`expected_counts` *Expected counts under indirect standardisation*

Description

The count each area would have if it experienced the overall rate in every stratum, given its own composition. Comparing observed against this rather than against a raw rate removes the part of the difference that is explained by who the area holds.

Usage

```
expected_counts(counts, population, area, strata = NULL)
```

Arguments

`counts`	Observed counts.
`population`	Population at risk, the same length as `counts`.
`area`	Area label for each row.
`strata`	Optional stratum label for each row – an age band, a gender, or their interaction. With strata the standardisation is indirect in the usual sense: the overall stratum-specific rates are applied to each area's own stratum populations.

Details

Without strata the expected count is just the area's population times the overall rate, which adjusts for size but not for composition. The two are worth distinguishing: a region holding disproportionately many young men will show an excess on the first and may show none on the second, and only the second is evidence about the region.

Value

A data frame with one row per area: `observed`, `population` and `expected`.

References

Lawson, A. B. *Using R for Bayesian Spatial and Spatio-Temporal Health Modeling*. Chapman and Hall/CRC. Chapter 1 sets out the convention used here: the expected counts come from applying the overall population rate to each area, and the standardised incidence ratio is the ratio of count to expected.

Hedderich, J. and Sachs, L. (2020). *Applied Statistics: Methods Using R*. Springer, on the distinction between indirect standardisation, which applies the reference's stratum-specific rates to the study population, and direct standardisation, which does the reverse.

See Also

`sir()`, `eb_rates()`, `funnel_limits()`

Examples

```
d <- data.frame(
  region = rep(c("North", "South", "East"), each = 2),
  age = rep(c("18 to 24", "25 to 49"), 3),
  n = c(30, 45, 12, 60, 8, 20),
  pop = c(1000, 6000, 900, 9000, 400, 3500)
)

# Adjusting for size only.
expected_counts(d$n, d$pop, d$region)

# Adjusting for size AND age composition, which is the comparison
# that says something about the region.
expected_counts(d$n, d$pop, d$region, strata = d$age)
```

`falsify_family` *Correct a family of falsification results for multiple testing*

Description

Takes the p-values from several `capsule_falsify()` runs – or a plain numeric vector – and reports which survive once the size of the family is accounted for.

Usage

```
falsify_family(x, method = c("holm", "bh", "bonferroni", "none"), alpha = 0.05)
```

Arguments

`x`	A named list of `bricklayer_falsification` objects, or a named numeric vector of p-values.
`method`	`"holm"` (the default), `"bh"`, `"bonferroni"` or `"none"`.
`alpha`	Threshold applied to the adjusted values.

Details

Running the permutation control over twenty statistics and reporting the one that came in under 0.05 is not a finding: at that family size roughly one spurious result is what chance produces. Which correction to use depends on the claim. Holm controls the probability of ANY false positive, which is what a claim about a specific statistic needs. Benjamini-Hochberg controls the expected PROPORTION of false positives among those declared, which is what a screening exercise needs; it is less conservative and says something weaker.

A permutation p-value cannot fall below `1 / (n + 1)`, so with a small number of permutations a whole family can be uncorrectable – every p sits at the floor. That is reported rather than hidden, because the alternative is a table of adjusted values that look like evidence of nothing in particular.

Value

A list of class `bricklayer_falsify_family`: a `results` data frame (`name`, `p`, `adjusted`, `survives`) , the `method`, the family size, and `at_floor`, the names whose p-value equals the smallest their permutation count could produce.

See Also

`capsule_falsify()`, `prereg_declare()`.

Examples

```
# four statistics, one of which is real
set.seed(1)
d <- data.frame(x = rnorm(150))
d$y <- 0.6 * d$x + rnorm(150)
d$a <- rnorm(150)
d$b <- rnorm(150)
fam <- list(
  real = capsule_falsify(d, function(z) cor(z$x, z$y),
                         treatment = "x", n = 199, seed = 1),
  noise_a = capsule_falsify(d, function(z) cor(z$a, z$y),
                            treatment = "a", n = 199, seed = 2),
  noise_b = capsule_falsify(d, function(z) cor(z$b, z$y),
                            treatment = "b", n = 199, seed = 3))
falsify_family(fam)

# the correction is what stops the smallest of several from being
# read as the finding
falsify_family(c(a = 0.01, b = 0.04, c = 0.2, d = 0.5))$results
```

`fips_key` *Assemble a standardised key from raw key material*

Description

Wraps key bytes that came from somewhere else – another implementation, a key store, a file written by an earlier session – in the object `capsule_sign()` and `capsule_verify()` expect. The lengths are checked against the parameter set, so material for the wrong scheme is refused here rather than producing a signature nothing can verify.

Usage

```
fips_key(scheme, public, secret = NULL)
```

Arguments

`scheme`	Scheme name, as in `fips_keygen()`.
`public`	Public key, hex or raw.
`secret`	Secret key, hex or raw. Omit for a verification-only key.

Details

The byte layouts are the standards' own, which is what makes this interoperable: an ML-DSA secret key is `rho || K || tr || s1 || s2 || t0` and an SLH-DSA one is `SK.seed || SK.prf || PK.seed || PK.root`, exactly as FIPS 204 and FIPS 205 encode them.

Value

A `bricklayer_fips_key` when `secret` is given, otherwise a `bricklayer_fips_public_key`.

See Also

`fips_keygen()`, `fips_sizes()`.

Examples

```
key <- fips_keygen("ML-DSA-44")
# the round trip through raw material changes nothing
again <- fips_key("ML-DSA-44", key$public, key$secret)
identical(again$secret, key$secret)

sig <- capsule_sign("m", again)
capsule_verify("m", sig, fips_key("ML-DSA-44", key$public))

# material of the wrong length is refused
try(fips_key("ML-DSA-65", key$public, key$secret))
```

`fips_keygen` *Generate a standardised post-quantum signing key*

Description

Generates a key for one of the NIST-standardised signature schemes: ML-DSA (FIPS 204) or SLH-DSA (FIPS 205). Both are implemented in this package, natively, with no system dependency; `pqc_backends()` lists the parameter sets.

Usage

```
fips_keygen(scheme = "ML-DSA-65", seed = NULL)

fips_public_key(key)
```

Arguments

`scheme`	Scheme name, one of `pqc_backends()` other than `"xmss-sha256"`. The default `"ML-DSA-65"` is the FIPS 204 middle security level.
`seed`	Optional raw vector of key-generation seed bytes, of the length `fips_sizes()` reports for the scheme. Supplying it makes the key reproducible, which is what the standards' test vectors need; the default draws from the operating system's CSPRNG.
`key`	A key from `fips_keygen()`.

Details

Unlike `pqc_keygen()` 's hash-based key, these are STATELESS: one key signs any number of messages, with no index to track and nothing to persist between signatures.

Which to pick. ML-DSA is small and fast and rests on a lattice assumption. SLH-DSA rests on nothing but the hash function, at the cost of a signature one to two orders of magnitude larger; its `s` parameter sets have small signatures and slow signing, its `f` sets the reverse, and its SHAKE and SHA-2 families are equally strong: pick SHA-2 where a validated SHA-2 implementation is what an auditor will ask about. `"ML-DSA-65"` is the sensible default.

Value

A list of class `bricklayer_fips_key`: `public`, `secret` (both hex), and `scheme`.

References

National Institute of Standards and Technology (2024). Module-Lattice-Based Digital Signature Standard. FIPS 204. doi:10.6028/NIST.FIPS.204

National Institute of Standards and Technology (2024). Stateless Hash-Based Digital Signature Standard. FIPS 205. doi:10.6028/NIST.FIPS.205

See Also

pqc_keygen() for the stateful hash-based key, capsule_sign() which accepts either, fips_sizes() for the byte lengths.

Examples

```
key <- fips_keygen("ML-DSA-65")
key$scheme

sig <- capsule_sign("a manifest digest", key)
capsule_verify("a manifest digest", sig, fips_public_key(key))

# A context string binds the signature to its purpose: the same
# message signed for one context does not verify under another.
sig2 <- capsule_sign("a manifest digest", key, context = "release")
capsule_verify("a manifest digest", sig2, key, context = "release")
capsule_verify("a manifest digest", sig2, key, context = "staging")
```

---

fips_mu *Sign an ML-DSA message digest computed elsewhere*

Description

The external-mu interface. fips_mu() reduces a message to the 64-byte value mu that is the only thing ML-DSA signing actually consumes; fips_sign_mu() signs that value and fips_verify_mu() checks it.

Usage

```
fips_mu(key, message, context = NULL, prehash = "none")

fips_sign_mu(key, mu, deterministic = FALSE)

fips_verify_mu(key, mu, signature)
```

Arguments

key	A key from fips_keygen() or fips_key(). An ML-DSA scheme: SLH-DSA has no external-mu interface, because its digest depends on per-signature randomness that the signer chooses.
message	Length-1 character vector or raw vector.
context	Optional context string, as in capsule_sign().
prehash	Pre-hash function, as in capsule_sign().
mu	The 64 raw bytes from fips_mu().
deterministic	As in capsule_sign().
signature	A signature from fips_sign_mu() or capsule_sign().

Details

The point is that the message need never reach the key. A large file can be reduced to mu on the machine that holds it and only mu handed to whatever holds the signing key – a smartcard, a remote signer, another process. mu is not a bare digest: it binds the public key (through `tr = H(pk)`) and the context string, so a mu computed under one key cannot be signed under another to any useful effect.

The resulting signature is an ordinary ML-DSA signature. `capsule_verify()` accepts it, given the same message and context.

Value

`fips_mu()` returns 64 raw bytes; `fips_sign_mu()` a `bricklayer_signature`; `fips_verify_mu()` a length-1 logical.

References

National Institute of Standards and Technology (2024). Module-Lattice-Based Digital Signature Standard. FIPS 204. doi:10.6028/NIST.FIPS.204

See Also

`capsule_sign()`, `fips_keygen()`.

Examples

```
key <- fips_keygen("ML-DSA-65")
mu <- fips_mu(key, "a manifest digest", context = "release")
length(mu)

sig <- fips_sign_mu(key, mu)
fips_verify_mu(key, mu, sig)

# the same signature verifies the ordinary way, from the message
capsule_verify("a manifest digest", sig, key, context = "release")

# mu is computable from the PUBLIC key alone, which is what lets the
# message stay on the machine that has it
identical(fips_mu(fips_public_key(key), "a manifest digest",
                  context = "release"), mu)
```

---

fips_sizes *Byte lengths of a standardised signature scheme*

Description

Reports the sizes fixed by a FIPS 204 or FIPS 205 parameter set, so a caller never has to hard-code them.

Usage

```
fips_sizes(scheme)
```

Arguments

scheme	Scheme name, as in fips_keygen().

Value

A named integer vector: public_key, secret_key, signature, seed and opt_rand, all in bytes. opt_rand is the per-signature randomness the scheme consumes.

See Also

fips_keygen(), pqc_backends().

Examples

```
fips_sizes("ML-DSA-65")
fips_sizes("SLH-DSA-SHAKE-128s")

# An SLH-DSA signature is far larger than an ML-DSA one at the same
# security level, which is the price of dropping the lattice
# assumption.
fips_sizes("SLH-DSA-SHAKE-128s")[["signature"]] >
  fips_sizes("ML-DSA-44")[["signature"]]
```

---

fiscal_year_label *Name a fiscal year by the years it spans*

Description

Ontario's inmate data keys on the fiscal year's END year, so 2023 is the fiscal year running from April 2022 to March 2023. This renders that as "2022/23".

Usage

```
fiscal_year_label(end_year, sep = "/", short = TRUE)
```

Arguments

end_year	The fiscal year's end year, as a number or a string.
sep	Separator between the two years.
short	Whether to abbreviate the second year to two digits.

Value

A character vector of labels.

Examples

```
fiscal_year_label(2019:2023)

fiscal_year_label(2023, short = FALSE)

fiscal_year_label(2023, sep = "-")
```

`frequency_table` *Frequency table for one column*

Description

Counts, percentages, and percentages of the non-missing values, with missing counted as its own row. The counterpart of `janitor::tabyl()`.

Usage

```
frequency_table(data, column = NULL, max_levels = 25L, sort = TRUE)
```

Arguments

`data`	A data frame, or a vector.
`column`	Column name, when `data` is a data frame.
`max_levels`	Maximum rows to return, most frequent first (default 25). The remainder are folded into one `"(other)"` row so the percentages still sum to 100.
`sort`	Sort by descending count (default `TRUE`) ; `FALSE` keeps the natural order of the values.

Details

Two percentage columns, because both questions get asked and conflating them is how missingness gets hidden: `pct` is the share of ALL rows, `pct_valid` the share of rows where the value is present. When a column is 40% missing those two differ enormously, and only the second describes the values that are actually there.

Value

A data frame of class `bricklayer_freq`: `value`, `n`, `pct`, `pct_valid`.

See Also

`profile_columns()` for every column at once.

Examples

```
df <- data.frame(grade = c("a", "b", "b", "c", NA, "b"),
                 stringsAsFactors = FALSE)
frequency_table(df, "grade")

# The two percentage columns differ exactly by the missingness.
frequency_table(df, "grade")[, c("pct", "pct_valid")]

# A vector works directly.
frequency_table(c(1, 1, 2, 3, 3, 3))

# Natural order rather than frequency order.
frequency_table(c("c", "a", "b", "a"), sort = FALSE)

# Long tails are folded so the percentages still total 100.
set.seed(1)
ft <- frequency_table(sample(letters, 500, TRUE), max_levels = 5)
ft
sum(ft$pct)
```

---

friendly_download *Download a File With Diagnostic Error Messages*

Description

Wraps `utils::download.file()` and, on failure, prints plain-language guidance for the most common academic and corporate network problems (rate limiting, TLS-inspection VPNs, DNS failures, timeouts, HTTP 403). Optionally retries from a Wayback Machine snapshot URL.

Usage

```
friendly_download(url, target_path, attempt_wayback = NULL)
```

Arguments

`url`	URL to download.
`target_path`	Destination path on disk.
`attempt_wayback`	Wayback Machine snapshot URL tried as a fallback if the primary download fails. When `NULL` (the default) a snapshot is resolved automatically via `wayback_snapshot_url()`; pass an explicit URL to override the lookup, or `""` to disable the fallback entirely.

Value

`TRUE` if either the primary download or the Wayback fallback succeeds, otherwise `FALSE`.

Examples

```
ok <- friendly_download("https://cloud.r-project.org/",
                        tempfile(fileext = ".html"),
                        attempt_wayback = "")  # disable the fallback
ok
```

`funnel_limits` *Funnel-plot control limits*

Description

The limits within which an area's ratio would fall, given its expected count, if it were no different from the overall experience. A funnel plot is the alternative to a league table: it shows directly that a small area's ratio can wander far from one without meaning anything.

Usage

```
funnel_limits(expected, target = 1, levels = c(0.95, 0.998))
```

Arguments

`expected`	Expected counts to compute limits at.
`target`	The ratio the limits are centred on. One is the overall experience.
`levels`	Two-sided coverage levels for the limit pairs.

Details

The limits are exact Poisson quantiles divided by the expected count, so they are the discrete counterpart of the usual normal funnel and stay correct at the small expected counts where the normal version goes below zero.

Value

A data frame of `expected`, `level`, `lower` and `upper` on the ratio scale.

References

Advanced Statistics in Criminology and Criminal Justice discusses the funnel plot as the display of the relationship between an estimate and the sample size behind it.

Lawson, A. B. *Using R for Bayesian Spatial and Spatio-Temporal Health Modeling*. Chapman and Hall/CRC, on the Poisson counts these limits are built from.

See Also

`sir()`, `eb_rates()`

Examples

```
# The funnel narrows as the expected count grows, which is the whole
# point: a ratio of 2 means nothing at an expected count of 2 and a
# great deal at an expected count of 200.
funnel_limits(c(2, 20, 200))
```

`hill_tail_index` *Tail index of a heavy-tailed count*

Description

The Clauset-Shalizi-Newman maximum-likelihood estimator for a discrete power law above a threshold. An exponent near 2 or below means the mean is barely defined and the observed maximum is not informative about the next one, which is the substantive point when a few units dominate a total.

Usage

```
hill_tail_index(
  x,
  x_min = NULL,
  discrete = TRUE,
  approx = FALSE,
  min_tail = 3L
)
```

Arguments

`x`	Positive values, one per unit.
`x_min`	Threshold above which the power law is fitted. A power law is a statement about the tail, so a threshold is required; the default takes the value that leaves at least 50 observations, or the minimum if the data are smaller than that.
`discrete`	Whether the quantity is integer-valued. A count is, and then the likelihood maximised is the zeta distribution's, whose normalising constant is a Hurwitz zeta.
`approx`	For discrete data, whether to use the closed-form continuity-corrected estimator instead of maximising the exact likelihood. It is much faster and much worse: the correction is an asymptotic approximation in `x_min`, and at `x_min` = 1 – where administrative counts start – it returns about 2.0 from data generated with an exponent of 2.5. Off by default for that reason.
`min_tail`	Fewest tail observations for which an estimate is reported at all. Below it there is nothing to estimate from and `alpha` is `NA`.

Value

A list with alpha, its standard error, x_min, n_tail, ks and reliable. ks is the Kolmogorov-Smirnov distance between the fitted tail and the data: a large value means the tail is not a power law, whatever alpha came out as. reliable is FALSE when fewer than 50 observations lie in the tail, which is the sample size Clauset, Shalizi and Newman give as the point below which the estimate should not be leaned on – it is reported rather than enforced, because the right response to a short tail is a wider interval, not a refusal.

References

Clauset, A., Shalizi, C. R. and Newman, M. E. J. (2009). Power-law distributions in empirical data. *SIAM Review* 51(4), 661-703. The estimator and its threshold guidance are theirs; the continuity correction they give in closed form is an asymptotic approximation in x_min, which is why the exact likelihood is maximised here instead. (Not in the local corpus; cited from the published paper.)

Examples

```
# A continuous Pareto tail with exponent 2.5.
set.seed(1)
x <- (1 - stats::runif(5000))^(-1 / 1.5)
round(hill_tail_index(x, x_min = 1, discrete = FALSE)$alpha, 2)

# A discrete power law, where the exact likelihood is needed: the
# closed-form correction is badly biased at a threshold of one.
k <- 1:10000
p <- k^(-2.5) / sum(k^(-2.5))
set.seed(2)
z <- sample(k, 5000, replace = TRUE, prob = p)
c(exact = round(hill_tail_index(z, x_min = 1)$alpha, 2),
  approx = round(hill_tail_index(z, x_min = 1, approx = TRUE)$alpha, 2))

# A short tail still returns an estimate, marked as not to be leaned
# on, and with a standard error that says the same thing.
short <- hill_tail_index(c(3, 4, 5, 9), x_min = 3)
c(alpha = round(short$alpha, 2), n = short$n_tail,
  reliable = short$reliable)

# Below `min_tail` there is nothing to estimate from.
hill_tail_index(c(3, 4), x_min = 3)$alpha
```

---

hurwitz_zeta *Hurwitz zeta function*

Description

zeta(s, q) = sum over k >= 0 of (q + k)^-s, by Euler-Maclaurin. It is the normalising constant of the discrete power law truncated below at q, which is why it is here; zeta(s, 1) is the Riemann zeta.

Usage

```
hurwitz_zeta(s, q = 1)
```

Arguments

s	Exponent, which must exceed 1 for the series to converge.
q	Lower limit, which must be positive.

Value

A numeric vector the length of s.

References

The Euler-Maclaurin expansion used here – direct terms to q + N, then the integral tail, then the Bernoulli-number corrections – is the standard evaluation; see Abramowitz, M. and Stegun, I. A. *Handbook of Mathematical Functions*, Sec. 23.2. (Not in the local corpus; cited from the published reference. The implementation is checked against pi^2/6, pi^4/90, Apery's constant and the shift identity, which is stronger evidence than the citation.)

Examples

```
# The Riemann zeta at even integers has a closed form.
c(hurwitz_zeta(2), pi^2 / 6)
c(hurwitz_zeta(4), pi^4 / 90)

# Apery's constant.
hurwitz_zeta(3)

# Shifting the lower limit removes exactly the leading term.
hurwitz_zeta(2.5, 3) - hurwitz_zeta(2.5, 4)
3^-2.5

# Outside the domain of convergence there is no value to return.
hurwitz_zeta(1)
```

---

infer_schema *Infer a pinnable schema from a data frame*

Description

Derives the schema validate_schema() consumes from data you already trust, so a capsule can be pinned without writing one by hand.

Usage

```
infer_schema(data, slack = 0.1, max_levels = 50L)
```

Arguments

data	A data frame to learn from.
slack	Fractional headroom added to row counts, numeric ranges and missingness rates (default 0.1, i.e. 10%). 0 pins exactly to what was observed, which will reject almost any re-release.
max_levels	Maximum distinct values for a categorical column to have its value set recorded (default 50). Above this the column is treated as free text and no value set is pinned.

Details

What it records: the column names and their types, row-count bounds with slack either side, the observed value set for every low-cardinality categorical column, the observed range of every numeric column widened by slack, and the observed missingness rate per column with headroom.

Value

A list with expected_columns, expected_types, structural_invariants, expected_value_sets, numeric_ranges and max_missing_fraction, of class bricklayer_schema. Wrap it as list(schema = <this>) to hand to validate_schema().

This is a starting point, not an oracle

An inferred schema describes ONE extract. It cannot know that a category which happens not to occur is nonetheless legal, or that a range is a physical bound rather than an accident of this sample. Read what it produces and edit it before committing – the value is in not starting from a blank file, not in trusting the output blindly.

See Also

validate_schema(), capsule_drift() for the distributional check the schema cannot make.

Examples

```
set.seed(1)
df <- data.frame(
  id = 1:100,
  score = stats::runif(100, 0, 10),
  grade = sample(c("a", "b", "c"), 100, TRUE),
  note = paste0("free text ", 1:100),
  stringsAsFactors = FALSE
)

sch <- infer_schema(df)
sch

# The categorical column has its levels pinned; the free-text one does
# not, because it exceeds max_levels.
sch$expected_value_sets
```

```
# It validates the data it was learned from.
length(validate_schema(df, list(schema = sch)))

# And catches a column that has gone missing, or a new category.
length(validate_schema(df[, -3], list(schema = sch))) > 0
bad <- df; bad$grade[1] <- "z"
length(validate_schema(bad, list(schema = sch))) > 0
```

`inline_hist` *Inline histogram for a numeric vector*

Description

A one-line sketch of a column's distribution, as block characters (or ASCII where the console cannot render them). The point is density of information: a mean and a standard deviation cannot tell you a column is bimodal, and this can, in the width of a table cell.

Usage

```
inline_hist(x, bins = 10L)
```

Arguments

`x`	Numeric vector.
`bins`	Number of bins (default 10).

Value

A length-1 character vector of `bins` characters.

See Also

`profile_columns()`, which includes one per numeric column.

Examples

```
set.seed(1)
# A symmetric distribution peaks in the middle.
inline_hist(stats::rnorm(1000))

# A skewed one leans left.
inline_hist(stats::rexp(1000))

# Bimodality is visible here and in no single summary number.
inline_hist(c(stats::rnorm(500, -3), stats::rnorm(500, 3)))

# A constant column has no spread to show.
inline_hist(rep(5, 10))
```

kem_decapsulate *Recover a shared secret from an ML-KEM ciphertext*

Description

Returns the 32-byte shared secret the ciphertext carries, as hex.

Usage

```
kem_decapsulate(key, ciphertext)
```

Arguments

key	A key from kem_keygen(), with its secret half.
ciphertext	Ciphertext from kem_encapsulate(), hex or raw.

Details

There is no failure path, and that is deliberate. A ciphertext that was not produced by a correct encapsulation under this key yields a shared secret derived from a value held only inside the decapsulation key – so it is a real secret, just not the sender's. Whoever sent it learns nothing about whether it was accepted, which is what closes off a chosen-ciphertext attack. The consequence for a caller: a mismatch between the two sides' secrets is the signal that something was wrong, not an error from this function.

Value

64 hex characters: the 32-byte shared secret.

See Also

kem_encapsulate().

Examples

```
key <- kem_keygen(512)
sent <- kem_encapsulate(kem_public_key(key))
identical(kem_decapsulate(key, sent$ciphertext), sent$shared)

# a tampered ciphertext returns a secret, and it is the wrong one
bad <- sent$ciphertext
substring(bad, 3L, 4L) <- "ff"
other <- kem_decapsulate(key, bad)
nchar(other) == 64L
identical(other, sent$shared)
```

`kem_encapsulate` *Encapsulate a shared secret under an ML-KEM key*

Description

Produces a ciphertext and the 32-byte shared secret it carries. Only the public key is needed, which is the point: the sender never holds anything the recipient has to trust them with.

Usage

```
kem_encapsulate(key, m = NULL)
```

Arguments

`key`	A key or public key from `kem_keygen()` / `kem_public_key()`.
`m`	Optional raw vector of 32 bytes of encapsulation randomness. Supplying it makes the operation reproducible, which is what the standard's test vectors need; the default draws from the operating system's CSPRNG. Reusing it across encapsulations to the same key reuses the shared secret, so supply it only deliberately.

Value

A list of class `bricklayer_kem_capsule`: `ciphertext` and `shared` (both hex), and `level`.

See Also

`kem_decapsulate()`, `kem_keygen()`.

Examples

```
key <- kem_keygen(512)
a <- kem_encapsulate(key)
nchar(a$ciphertext) / 2 == kem_sizes(512)[["ciphertext"]]

# two encapsulations to one key give different secrets
b <- kem_encapsulate(key)
identical(a$shared, b$shared)

# both decapsulate correctly
identical(kem_decapsulate(key, a$ciphertext), a$shared)
identical(kem_decapsulate(key, b$ciphertext), b$shared)
```

kem_keygen *Generate an ML-KEM key pair*

Description

Generates a key for ML-KEM (FIPS 203), the standardised post-quantum key encapsulation mechanism. It is implemented in this package, with no system dependency.

Usage

```
kem_keygen(level = 768L, seed = NULL)

kem_public_key(key)
```

Arguments

`level`	Security level: 512, 768 (the default) or 1024. 768 is the level NIST and the IETF have settled on for general use.
`seed`	Optional raw vector of 64 seed bytes (`d || z`) . Supplying it makes the key reproducible, which is what the standard's test vectors need; the default draws from the operating system's CSPRNG.
`key`	A key from `kem_keygen()`.

Details

A KEM is not a signature scheme and not a cipher. Encapsulation produces two things: a ciphertext to send, and a 32-byte shared secret to keep. The holder of the decapsulation key recovers the same secret from the ciphertext. What either side then does with that secret – feed it to a KDF, key an AEAD – is outside the mechanism.

Value

A list of class `bricklayer_kem_key`: `public`, `secret` (both hex), and `level`.

References

National Institute of Standards and Technology (2024). Module-Lattice-Based Key-Encapsulation Mechanism Standard. FIPS 203. doi:10.6028/NIST.FIPS.203

See Also

`kem_encapsulate()`, `kem_decapsulate()`, `kem_sizes()`, `fips_keygen()` for the signature schemes.

Examples

```
key <- kem_keygen(768)
pub <- kem_public_key(key)

# the sender holds only the public key
sent <- kem_encapsulate(pub)
# the recipient recovers the same secret from the ciphertext
got <- kem_decapsulate(key, sent$ciphertext)
identical(sent$shared, got)

# a corrupted ciphertext yields a DIFFERENT secret, not an error
bad <- sent$ciphertext
substring(bad, 1L, 2L) <- "00"
identical(kem_decapsulate(key, bad), got)
```

---

kem_sizes *Byte lengths of an ML-KEM parameter set*

Description

Reports the sizes FIPS 203 fixes, so a caller never has to hard-code them.

Usage

```
kem_sizes(level)
```

Arguments

level	Security level: 512, 768 or 1024.

Value

A named integer vector: `encapsulation_key`, `decapsulation_key`, `ciphertext`, `seed` and `shared_secret`, all in bytes.

See Also

`kem_keygen()`.

Examples

```
kem_sizes(768)

# the shared secret is 32 bytes at every level: the level buys
# security margin, not a longer secret
vapply(c(512, 768, 1024),
       function(l) kem_sizes(l)[["shared_secret"]], integer(1))
```

load_provenance *Load a Pinned Data-Provenance Record*

Description

Reads a `data_provenance.json` file describing a project's pinned data source: the CKAN endpoint, resource name pattern, expected SHA256, Wayback snapshot, schema, and synthetic-data recipe.

Usage

```
load_provenance(path)
```

Arguments

`path`	Path to the provenance JSON file.

Value

The parsed provenance as a nested list, read with the package's own JSON codec (`bricklayer_json_from_json()`, unsimplified), or `NULL` if the file does not exist.

Examples

```
prov_file <- tempfile(fileext = ".json")
writeLines('{"dataset": {"title": "demo"}, "sha256": "abc"}', prov_file)
prov <- load_provenance(prov_file)
prov$dataset$title
load_provenance(file.path(tempdir(), "no-such-file.json"))  # NULL
```

mahalanobis_outliers *Multivariate outliers by Mahalanobis distance*

Description

Ranks rows by how far they sit from the centre of the data ONCE THE CORRELATIONS ARE ACCOUNTED FOR, which is what a per-column check cannot do: a row can be unremarkable on every variable separately and still be impossible jointly – a person 1.5 m tall weighing 140 kg is inside both marginal ranges and outside the cloud.

Usage

```
mahalanobis_outliers(data, alpha = 0.001, robust = TRUE)
```

Arguments

data	A data frame or numeric matrix; non-numeric columns are dropped. Rows with any missing value are skipped, and reported as NA.
alpha	Significance level for the outlier flag (default 0.001, deliberately strict: at 0.05 one row in twenty is flagged by construction).
robust	Use the median/MAD centre and scale (default TRUE) .

Details

Under multivariate normality the squared distance is chi-square on p degrees of freedom, which gives the p-value and the cut-off.

robust = TRUE centres on the coordinate-wise median and scales by a MAD-based covariance instead of the mean and sample covariance. This matters more than it sounds: outliers inflate the very covariance used to judge them, so with several of them the classical distance hides exactly the rows it is meant to find (the masking effect).

Exactly collinear columns are refused rather than repaired: the distance is undefined there, and flooring the covariance's eigenvalues to make it invertible would return numbers governed by the floor rather than by the data. Drop the redundant column first – top_correlations() will identify it.

Value

A data frame of class bricklayer_outliers with row, distance (the square root of the squared Mahalanobis distance), p_value and outlier, ordered by descending distance.

References

Mahalanobis PC (1936). On the generalised distance in statistics. *Proceedings of the National Institute of Sciences of India* 2(1), 49–55.

See Also

core_tukey_fences() for the per-column version, rule_within_n_mads() to turn it into a validation rule.

Examples

```
set.seed(1)
n <- 200
df <- data.frame(height = stats::rnorm(n, 170, 10))
df$weight <- df$height * 0.5 + stats::rnorm(n, 0, 5)

# A row that is ordinary on each variable but impossible jointly.
df[1, ] <- list(height = 150, weight = 140)

out <- mahalanobis_outliers(df)
head(out, 3)

# It is flagged, even though neither value is a marginal outlier.
```

```
out$row[1] == 1
range(df$height)
range(df$weight)

# The classical version can be fooled by the outliers it should find.
mahalanobis_outliers(df, robust = FALSE)$distance[1] <
  mahalanobis_outliers(df, robust = TRUE)$distance[1]

# A duplicated column has no distance defined, and is refused.
dup <- df
dup$height2 <- dup$height
try(mahalanobis_outliers(dup))
```

`make_manifest` *Construct a Reproducibility Manifest*

Description

Creates an empty manifest object that accumulates cross-check entries via `record()` and is later serialized with `write_manifest_json()`.

Usage

```
make_manifest(meta, environment = TRUE)
```

Arguments

`meta`	A named list of run metadata (e.g. `project`, `author`, `run_at`, `synthetic`) .
`environment`	Logical; when `TRUE` (the default) the manifest also records the analysis environment via `capture_environment()` (R version, platform, OS, UTC timestamp, loaded package versions).

Value

A manifest list with elements `meta`, an empty `results` list, and (when requested) `environment`.

Examples

```
# Minimal manifest, no environment capture.
man <- make_manifest(list(project = "demo-study", author = "A. Author"),
                     environment = FALSE)
names(man)          # "meta" "results"
man$meta$project

# With environment = TRUE it also records R version / platform / packages.
full <- make_manifest(list(project = "demo"), environment = TRUE)
names(full)         # adds "environment"
full$environment$r_version
```

make_synthetic_column *Generate One Synthetic Column From a Spec*

Description

Builds a single synthetic data column according to a column spec drawn from a provenance synthetic recipe. Supported `type` s are `"sample"` (categorical, optionally weighted), `"bernoulli"` (two-label draw with optional per-row base rate), `"poisson"` (counts with a floor), `"id_pattern"` (templated IDs, optionally per-year sequenced), and `"sequence"` (a running integer sequence).

Usage

```
make_synthetic_column(spec, n, ctx = list(), base_p = NULL)
```

Arguments

`spec`	A list describing the column; recognised fields depend on `spec$type` (e.g. `values`, `weights`, `p`, `p_with_baserate`, `labels`, `lambda`, `min`, `pattern`, `year_col`, `from`) .
`n`	Number of values to generate.
`ctx`	Named list of already-generated columns, letting later columns (such as `id_pattern` with a `year_col`) reference earlier ones. Defaults to an empty list.
`base_p`	Optional numeric vector of per-row latent propensities used by the `"bernoulli"` type to add row-level variation.

Value

A vector of length `n` for the requested column type. Errors on an unknown `type`.

Examples

```
set.seed(1)
# "sample": categorical draw, optionally weighted.
make_synthetic_column(list(type = "sample", values = list("a", "b"),
                           weights = list(0.7, 0.3)), 5)

# "bernoulli": two-label draw at probability p.
make_synthetic_column(list(type = "bernoulli", p = 0.5,
                           labels = list("Yes", "No")), 5)

# "poisson": counts with a floor via `min`.
make_synthetic_column(list(type = "poisson", lambda = 3, min = 1), 5)

# "id_pattern": templated IDs (the {seq:05d} token is zero-padded).
make_synthetic_column(list(type = "id_pattern",
                           pattern = "case-{seq:05d}"), 3)

# "sequence": a running integer sequence from `from`.
```

```
make_synthetic_column(list(type = "sequence", from = 100), 4)

# An unknown type errors.
try(make_synthetic_column(list(type = "nope"), 3))
```

`make_synthetic_csv` *Generate a Synthetic CSV From a Schema Recipe*

Description

Generates a reproducible synthetic data set from the `schema$synthetic_recipe` block of a provenance object and writes it to a CSV. Columns are produced in declaration order so later columns can reference earlier ones, a shared per-row latent propensity drives any Bernoulli columns, and an optional row-replication block expands per-person rows.

Usage

```
make_synthetic_csv(schema, out_path, n_rows = NULL, seed = NULL)
```

Arguments

`schema`	The synthetic recipe (a list with `columns`, and optional `n_rows`, `seed`, and `row_replication`).
`out_path`	Path where the CSV is written.
`n_rows`	Number of rows (persons, if replicating) to generate. Defaults to `schema$n_rows`, then 50000.
`seed`	Random seed for reproducibility. Defaults to `schema$seed`, then 91735246.

Value

Invisibly, a list with `path`, `rows` (rows written), and `seed` used.

Examples

```
recipe <- list(
  n_rows = 20, seed = 42,
  columns = list(
    year   = list(type = "sample", values = list(2024, 2025)),
    alert  = list(type = "bernoulli", p = 0.2),
    visits = list(type = "poisson", lambda = 3, min = 1),
    id     = list(type = "id_pattern", pattern = "p-{seq:05d}")
  )
)
out <- tempfile(fileext = ".csv")
res <- make_synthetic_csv(recipe, out)
res$rows                          # 20
res$seed                          # 42 (reproducible)
```

```
# The written CSV round-trips and has the declared columns.
df <- utils::read.csv(out)
dim(df)
names(df)

# `n_rows` overrides the recipe's own row count.
make_synthetic_csv(recipe, tempfile(fileext = ".csv"), n_rows = 5)$rows
```

`manifest_canonical` *The canonical serialisation of a manifest, and its digest*

Description

`manifest_canonical()` renders a manifest as one line of JSON with every object's keys in sorted order and every number at full double precision. `manifest_digest()` is the SHA-256 of those bytes.

Usage

```
manifest_canonical(manifest)

manifest_digest(manifest)
```

Arguments

`manifest`	A manifest, as from `make_manifest()`.

Details

Why this is needed. Two manifests that record the same thing can easily differ as bytes: R lists keep insertion order, so building `meta` before `results` or the other way round gives different JSON, and a signature over the JSON would then depend on the order a script happened to assemble the list. Sorting the keys removes that. The precision matters for a different reason: the default JSON writer emits four significant digits, which is right for a human-readable report and wrong for a record something will later be checked against, because `1/3` comes back as `0.3333` and no recomputation can match it.

Sign `manifest_digest()`, not the pretty JSON. The digest is stable across the assembly order, across `pretty`, and across a round trip through a file.

Full precision means full precision on every platform, which took more than writing enough digits. Seventeen significant digits recover any double exactly, but only through a reader that converts decimal to binary with correct rounding, and not every C library does – macOS arm64 reads the correct decimal for the largest double as infinity. So this package converts decimals itself rather than asking the platform, in integer arithmetic with a remainder that decides the rounding. A manifest written on one machine reads back bit-identically on another, and a caller does nothing to get that.

Value

`manifest_canonical()` a length-1 character vector; `manifest_digest()` 64 hex characters.

See Also

make_manifest(), write_manifest_json(), capsule_attest().

Examples

```
a <- make_manifest(list(b = 2, a = 1), environment = FALSE)
b <- make_manifest(list(a = 1, b = 2), environment = FALSE)
# the same content in a different order has the same digest
identical(manifest_digest(a), manifest_digest(b))

# full precision, so a recorded number can be checked later
m <- make_manifest(list(x = 1/3), environment = FALSE)
grepl("0.33333333333333331", manifest_canonical(m), fixed = TRUE)
```

---

manifest_recompute *Recompute a manifest's recorded statistics against the data*

Description

Re-evaluates named statistics against data and compares each to what the manifest recorded. This is the check verify_capsule() cannot make: that one confirms the record is internally consistent, which an edited-at-the-time manifest also is.

Usage

```
manifest_recompute(manifest, data, statistics, tol = NULL)
```

Arguments

manifest	A manifest, as from make_manifest() and record().
data	The data the statistics are computed from.
statistics	Named list of functions of data. Names must match the recorded result names.
tol	Optional numeric tolerance overriding each result's own recorded tol. Supply it to check more strictly than the original run did.

Details

Every recorded result that was NOT recomputed is reported too, as unchecked. An analysis that recorded twenty statistics and re-derives three has seventeen it has not re-derived, and a report that quietly omitted them would read as a clean bill of health.

Value

A list of class bricklayer_recompute: ok, a results data frame with one row per recomputed statistic (name, recorded, recomputed, delta, tol, status) , and unchecked, the names recorded but not recomputed.

See Also

verify_capsule(), capsule_falsify(), record().

Examples

```
d <- data.frame(x = 1:10)
m <- make_manifest(list(dataset = "demo"), environment = FALSE)
m <- record(m, "mean_x", observed = mean(d$x), expected = 5.5)
m <- record(m, "n", observed = nrow(d), expected = 10)

# recomputing both reproduces them
res <- manifest_recompute(m, d, list(mean_x = function(z) mean(z$x),
                                     n = function(z) nrow(z)))
res$ok
res$results[, c("name", "recorded", "recomputed", "status")]

# recomputing one leaves the other reported as unchecked
manifest_recompute(m, d, list(n = function(z) nrow(z)))$unchecked

# and data that no longer matches the record is caught
manifest_recompute(m, data.frame(x = 1:11),
                   list(n = function(z) nrow(z)))$ok
```

---

manifest_record_seed *Record and restore the random number generator state*

Description

manifest_record_seed() stores the generator's kind and its full state in the manifest; manifest_restore_seed() puts both back. Together they let a run that used randomness be repeated exactly.

Usage

```
manifest_record_seed(manifest)

manifest_restore_seed(manifest)
```

Arguments

manifest	A manifest, as from make_manifest().

Details

Why the state and not just a seed. set.seed(1) is reproducible only if everything before it is too: a single extra draw anywhere upstream shifts every subsequent value. Recording .Random.seed as it stood pins the actual position in the stream, and recording RNGkind() alongside it pins what that position means – R has changed its default sample() algorithm before, and a seed replayed under a different kind gives different numbers with no warning.

This is opt-in because the state is 626 integers, which is a lot of manifest for an analysis with no randomness in it.

Value

`manifest_record_seed()` the manifest with an `rng` element; `manifest_restore_seed()` the manifest, invisibly, having set the generator.

See Also

`capture_environment()`, `capsule_falsify()`.

Examples

```
set.seed(42)
m <- manifest_record_seed(make_manifest(list(a = 1),
                                        environment = FALSE))
first <- runif(3)

# any amount of other work can happen in between
invisible(runif(1000))

manifest_restore_seed(m)
identical(runif(3), first)
```

---

mcar_test *Little's test for data missing completely at random*

Description

Tests the MCAR assumption: that whether a value is missing is unrelated to any value in the data, observed or not.

Usage

```
mcar_test(data, max_iter = 500L, tol = 1e-07)
```

Arguments

`data`	A data frame or numeric matrix. Non-numeric columns are dropped with a warning, since the statistic is defined on moments.
`max_iter`	Maximum EM iterations (default 500).
`tol`	Convergence tolerance on the largest parameter change (default 1e-7).

Details

The assumption matters because it is what licenses the easy options. Dropping incomplete rows is unbiased under MCAR and biased otherwise; mean imputation understates variance under MCAR and distorts the centre as well otherwise. A small p-value here says the easy options are not available.

Value

A list of class `bricklayer_mcar`: `statistic`, `df`, `p_value`, `n_patterns`, `n_used`, `n_vars`, `iterations`, `mu`, `sigma`, and `note` (a caveat when the test is degenerate). With complete data `df` is 0 and `p_value` is `NA` – there is nothing to test.

How it works

Rows are grouped by their pattern of missingness. If missingness is unrelated to the values, then each pattern's observed-variable means should agree with the overall estimates, up to sampling error. The statistic sums those disagreements,

$$d^2 = \sum_j n_j (\bar{x}_j - \hat{\mu}_j)' \hat{\Sigma}_j^{-1} (\bar{x}_j - \hat{\mu}_j),$$

over the variables observed in pattern j, and is compared with a chi-square distribution on $\sum_j p_j - p$ degrees of freedom.

The overall $\hat{\mu}$ and $\hat{\Sigma}$ are the MAXIMUM-LIKELIHOOD estimates under multivariate normality WITH the missing data, obtained by expectation-maximisation – not the complete-case estimates, which would already embed the bias the test is looking for.

What it cannot do

A large p-value is NOT evidence that the data are MCAR; it is a failure to detect a departure, and the test has little power on small samples or with many patterns. The test also assumes multivariate normality, so on markedly non-normal columns a rejection may be telling you about the distribution rather than the missingness.

Neither this test nor any other can distinguish missing-at-random from missing-NOT-at-random, because that distinction depends on the values that were never observed. Only knowledge of how the data were collected settles it.

Collinear or constant columns are refused rather than worked around: the likelihood is degenerate there, and the EM step's eigenvalue floor would otherwise return a statistic governed by that floor instead of by the data.

References

Little RJA (1988). A test of missing completely at random for multivariate data with missing values. *Journal of the American Statistical Association* 83(404), 1198–1202. doi:10.1080/01621459.1988.10478722

Dempster AP, Laird NM, Rubin DB (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society B* 39(1), 1–38.

See Also

`missingness_pattern()` for the patterns themselves, `missingness_summary()` for the rates.

Examples

```
set.seed(1)
n <- 300
x <- stats::rnorm(n)
y <- x + stats::rnorm(n)

# Missing completely at random: a coin flip decides, so the test
# should not reject.
mcar <- data.frame(x = x, y = y)
mcar$y[sample(n, 90)] <- NA
mcar_test(mcar)

# Missing depending on the OTHER, observed variable: not MCAR, and
# detectable, because the pattern's mean of x is shifted.
mar <- data.frame(x = x, y = y)
mar$y[x > 0.4] <- NA
mcar_test(mar)

# Complete data has one pattern and nothing to test.
mcar_test(data.frame(a = x, b = y))$df

# A duplicated column makes the likelihood degenerate, and is refused.
dup <- mcar
dup$x2 <- dup$x
try(mcar_test(dup))

# The EM estimates are the ML ones: with no missingness they are the
# column means and the ML (1/n) covariance.
fit <- mcar_test(data.frame(a = x, b = y))
all.equal(fit$mu, c(mean(x), mean(y)), check.attributes = FALSE)
```

---

missingness_map *Text map of where the missing values are*

Description

Draws the missingness of a whole table as a grid, one character per cell block – a console counterpart of `visdat::vis_miss()` that needs no graphics device, so it works over SSH, in a log, and inside a capsule's plain-text summary.

Usage

```
missingness_map(data, height = 20L, width = 12L)
```

Arguments

`data`	A data frame.
`height`	Maximum rows in the map (default 20).
`width`	Maximum characters per column label (default 12).

Details

Rows are binned so the map fits `height` lines; a block is drawn at the shade its missing proportion falls in. Seeing the table at once is the point: a diagonal band, a block of rows, or a single ragged column are all instantly recognisable shapes that a column of percentages is not.

Value

A character vector of the map's lines, invisibly; printed as a side effect.

See Also

`missingness_pattern()`, `missing_runs()`

Examples

```
set.seed(1)
df <- data.frame(
  complete = 1:100,
  block = c(rep(NA, 30), 31:100),
  scattered = ifelse(stats::runif(100) < 0.3, NA, 1),
  mostly_gone = c(1:10, rep(NA, 90))
)
missingness_map(df)
```

`missingness_pattern` *Which columns are missing together*

Description

Counts the distinct PATTERNS of missingness across rows, rather than the per-column rates `profile_columns()` reports.

Usage

```
missingness_pattern(data, max_patterns = 20L)
```

Arguments

`data`	A data frame.
`max_patterns`	Maximum patterns to return, most frequent first (default 20).

Details

The distinction decides what to do about the gaps. Two columns each 30% missing at random need different handling from two columns 30% missing in THE SAME rows – the second is one structural gap (a join that failed, a form section nobody filled in) and often means those rows should be dropped or modelled separately, while the first does not. A per-column rate cannot tell the two apart; this can.

Value

A data frame of class `bricklayer_missingness`, one row per pattern: `pattern` (a string of `.` for present and `X` for missing, in column order), `n_rows`, `pct_rows`, `n_missing` (columns missing in that pattern), and `columns` (their names). Carries the column order as the `"columns"` attribute.

See Also

`profile_columns()` for per-column rates.

Examples

```
# Two columns missing in the SAME rows: one structural gap.
structural <- data.frame(
  id = 1:10,
  a = c(rep(NA, 3), 4:10),
  b = c(rep(NA, 3), 4:10)
)
missingness_pattern(structural)

# The same per-column rates, but missing independently.
scattered <- data.frame(
  id = 1:10,
  a = c(rep(NA, 3), 4:10),
  b = c(1:7, rep(NA, 3))
)
missingness_pattern(scattered)

# A complete frame has exactly one pattern.
missingness_pattern(data.frame(x = 1:3, y = 4:6))
```

`missingness_summary` *Missingness in one line per question*

Description

The scalar summaries of missingness: how much of the table is missing, how many rows are complete, and how many columns are wholly present.

Usage

```
missingness_summary(data)
```

Arguments

`data`	A data frame.

Value

A named numeric vector: `n_rows`, `n_cols`, `n_missing`, `pct_missing`, `n_complete_rows`, `pct_complete_rows`, `n_cols_any_missing`, `n_cols_all_missing`.

See Also

missingness_pattern() for which columns are missing together, profile_columns() for per-column rates.

Examples

```
df <- data.frame(a = c(1, NA, 3), b = c(NA, NA, 3), c = 1:3)
missingness_summary(df)

# A complete table is all zeros but for its dimensions.
missingness_summary(data.frame(x = 1:3, y = 4:6))
```

---

missing_runs *Runs of consecutive missing values*

Description

Finds the maximal stretches of consecutive NA in each column, with where each begins and how long it is.

Usage

```
missing_runs(data, min_run = 2L)
```

Arguments

data	A data frame.
min_run	Report only runs at least this long (default 2, since a run of 1 is an isolated gap).

Details

Row order carries meaning in a capsule far more often than people allow for – a time series, an ordered export, a paginated download – and a long unbroken run of missingness means something different from the same count scattered about. A run says an instrument was down, a page failed to fetch, or a period was never collected; scattered gaps say individual records failed. The rate cannot distinguish them.

Value

A data frame of class bricklayer_runs with column, start, end and length, longest first. Zero rows when there are no qualifying runs.

See Also

missingness_pattern() for which columns are missing together, missingness_summary() for the rates.

Examples

```
# One long outage and two isolated gaps, with the same total count.
df <- data.frame(
  outage = c(1, 2, NA, NA, NA, NA, 7, 8),
  scattered = c(1, NA, 3, 4, NA, 6, NA, NA)
)
sum(is.na(df$outage)) == sum(is.na(df$scattered))

missing_runs(df)

# Isolated gaps too, by lowering the threshold.
missing_runs(df, min_run = 1)

# A complete column has no runs.
missing_runs(data.frame(x = 1:5))
```

`morans_i` *Global Moran's I over a neighbour list*

Description

Spatial autocorrelation for an areal variable, with a permutation p-value. A neighbour list is required and is not invented: an administrative extract keyed on a region ships no geometry, and guessing adjacency would make the answer a property of the guess.

Usage

```
morans_i(x, neighbours, style = c("W", "B"), n_perm = 9999L)
```

Arguments

`x`	The variable, one value per area.
`neighbours`	Either a list with one integer vector of neighbour indices per area, or a square weight matrix.
`style`	`"W"` row-standardises the weights, so each area's neighbours carry a total weight of one; `"B"` leaves them binary. Row standardisation is the usual choice, and stops an area with many neighbours from dominating.
`n_perm`	Permutations for the null distribution.

Details

The expectation of I under the null is `-1/(n - 1)`, not zero, so a small negative I is what independence looks like in a small set of areas. The p-value comes from permuting the values over the areas, which needs no distributional assumption – and with a handful of regions no distributional assumption is safe.

Value

A list with `I`, its expectation under the null (`-1/(n-1)`, which is not zero), the permutation mean and standard deviation, a `z` score, `p_value`, and `W`, the total weight.

References

Moran, P. A. P. (1950). Notes on continuous stochastic phenomena. *Biometrika* 37(1/2), 17-23. (Not in the local corpus; cited from the published paper. The corpus does carry applied uses of the statistic, including Laniyonu (2017) on policing practices in gentrifying neighbourhoods, where it is used exactly as here – to establish that areal residuals are spatially dependent.)

Examples

```
# Six areas in a line, each adjacent to the next.
nb <- list(2L, c(1L, 3L), c(2L, 4L), c(3L, 5L), c(4L, 6L), 5L)

# A smooth gradient is strongly positively autocorrelated.
set.seed(1)
morans_i(c(1, 2, 3, 4, 5, 6), nb, n_perm = 999L)[c("I", "p_value")]

# An alternating pattern is negatively autocorrelated.
set.seed(1)
morans_i(c(1, 6, 1, 6, 1, 6), nb, n_perm = 999L)[c("I", "p_value")]

# And the null expectation is not zero.
-1 / (6 - 1)
```

---

oqs_keygen *Generate a standardised post-quantum signing key (deprecated name)*

Description

Kept so code written against the liboqs-backed version keeps working. The schemes are no longer reached through liboqs – they are implemented in this package – and the name no longer describes anything, so use `fips_keygen()` instead.

Usage

```
oqs_keygen(scheme = "ML-DSA-65")

oqs_public_key(key)
```

Arguments

`scheme`	Scheme name; see `fips_keygen()`.
`key`	A key from `oqs_keygen()`.

Value

As fips_keygen() and fips_public_key().

See Also

fips_keygen().

Examples

```
# Deprecated: use fips_keygen().
key <- suppressWarnings(oqs_keygen("ML-DSA-65"))
key$scheme
```

---

parse_bands *Parse banded category labels into numeric bounds*

Description

Recognises the forms that open-data publishers actually use: a bare number ("3") , a closed range ("18 to 24", "18-24", "18 - 24") , an open upper band ("50+", "Greater than 15", "over 15", "more than 15", "65 and over") , and an open lower band ("<18", "under 18", "less than 18") .

Usage

```
parse_bands(x, closed_upper = TRUE, integer_scale = TRUE)
```

Arguments

x	Character labels.
closed_upper	For an open upper band, whether the stated number is included. "50+" includes 50; "Greater than 15" does not, so its lower bound is 16 for integer data. Controlled per-label by the wording, and this argument only settles the ambiguous "+" form.
integer_scale	Whether the quantity is integer-valued, which decides whether "greater than 15" starts at 16 or just above 15.

Value

A data frame with label, lower, upper, open_lower and open_upper. An unparseable label gives NA bounds rather than a guess.

References

The forms recognised here are taken from the categories the Ontario Ministry of the Solicitor General publishes in its inmate datasets (the segregation and restrictive-confinement releases), where the placement-count and age categories are banded and the top band is open.

See Also

band_values(), band_sensitivity()

Examples

```
parse_bands(c("1", "2 to 5", "6 to 10", "Greater than 10"))

parse_bands(c("18 to 24", "25 to 49", "50+"))

# An unrecognised label is reported as unparsed, not guessed at.
parse_bands(c("3", "unknown", "not stated"))
```

---

period_days *Period length in days, from dates*

Description

Period length in days, from dates

Usage

```
period_days(from, to)
```

Arguments

from	Start of the period: a Date, or anything as.Date() accepts.
to	End of the period, inclusive.

Details

Inclusive of both ends, because a period running from the 1st to the 31st is thirty-one days of exposure, not thirty. Leap years need no special handling: the arithmetic is on dates, so 2024 comes out at 366 and 2023 at 365 without anyone choosing.

Value

The number of days in the period, for use as t.

See Also

adp()

Examples

```
period_days("2024-01-01", "2024-12-31")   # a leap year
period_days("2023-01-01", "2023-12-31")
period_days("2025-04-01", "2026-03-31")   # a fiscal year
```

pqc_backends *Available post-quantum signature schemes*

Description

Reports the signature schemes this package implements. The list is fixed, not probed: all of them are implemented in the package's own C++ and none depends on a system library, so a scheme available on one machine is available on every machine.

Usage

```
pqc_backends()
```

Details

`"xmss-sha256"` is the stateful hash-based scheme of `pqc_keygen()`. The rest are the NIST standards, taken with `fips_keygen()`: ML-DSA (FIPS 204) at all three parameter sets, and SLH-DSA (FIPS 205) at all twelve – six over SHAKE and six over SHA-2, which are different schemes and not merely different code paths.

Value

A character vector of scheme names, `"xmss-sha256"` first.

See Also

`fips_keygen()` for the standardised schemes, `pqc_keygen()` for the stateful one.

Examples

```
pqc_backends()

# Every scheme is present in every build.
all(c("xmss-sha256", "ML-DSA-65", "SLH-DSA-SHAKE-128s") %in%
    pqc_backends())
```

pqc_keygen *Generate a post-quantum signing key for capsule provenance*

Description

Builds an XMSS key pair: a Merkle tree over `2^height` Winternitz one-time keys, all derived from two 32-byte seeds. Security rests on SHA-256 alone – no lattice assumption, no elliptic curve, nothing Shor's algorithm breaks.

Usage

```
pqc_keygen(height = 10L, sk_seed = NULL, pub_seed = NULL, sk_prf = NULL)
```

Arguments

height	Tree height, 1 to 16 (default 10, i.e. 1024 signatures).
sk_seed, pub_seed, sk_prf	64-character hex seeds (32 bytes each) – the three secrets RFC 8391's private key carries. sk_seed derives the WOTS+ chains, pub_seed masks the hashes, and sk_prf keys the per-signature randomiser. Omit them and they are drawn from the operating system's CSPRNG via random_bytes(), which fails rather than falling back to R's reproducible generator. Supply them ONLY to reproduce a key deterministically in a test – a seed you can guess is a key you can forge.

Value

A list of class bricklayer_signing_key: root (the public verification value), pub_seed, sk_seed (SECRET), sk_prf (SECRET), height, next_index, capacity, and scheme.

A height-h key signs exactly 2^h messages

Each signature consumes one leaf, and **signing two different messages with the same leaf index breaks the scheme outright** – between two signatures at one index an adversary can forge a third message. capsule_sign() therefore tracks next_index and refuses to reuse one. Do not hand-edit that field, and do not copy a key to two machines that sign independently.

Key generation walks all 2^height leaves, so cost doubles with each unit of height. The default 10 gives 1024 signatures and takes a moment; heights above about 14 are slow enough to be worth avoiding unless the key really must last that long.

Key format

A key made before the RFC 8391 conformance work carries no sk_prf and cannot sign; capsule_sign() raises rather than producing a signature no other implementation could read. Generate a new one.

See Also

capsule_sign(), capsule_verify(), signing_public_key()

Examples

```
# A small key, to keep the example quick.
key <- pqc_keygen(height = 3)
key$capacity            # 8 signatures
key$next_index          # none used yet

# The public half is what a verifier needs; it carries no secret.
pub <- signing_public_key(key)
names(pub)
```

```
# Deterministic seeds reproduce the same key -- for tests only.
s1 <- paste(rep("11", 32), collapse = "")
s2 <- paste(rep("22", 32), collapse = "")
identical(pqc_keygen(3, s1, s2)$root, pqc_keygen(3, s1, s2)$root)
```

`prereg_declare` *Declare an analysis before running it*

Description

Records the statistics an analysis intends to report and what each is meant to show, so that what was actually reported can be compared against it afterwards. Seal it with `capsule_attest()` and the declaration acquires a date it cannot be moved off.

Usage

```
prereg_declare(hypotheses, note = NULL)

prereg_check(prereg, reported)
```

Arguments

`hypotheses`	Named character vector: one claim per statistic, named by the statistic's name as it will be recorded.
`note`	Optional free text – the design, the data source, what would count as a refutation.
`prereg`	A declaration from `prereg_declare()`.
`reported`	Character vector of the statistic names actually reported, or a manifest from which they are taken.

Details

The comparison `prereg_check()` makes is asymmetric on purpose, because the two ways of departing from a plan are different failures:

- a declared statistic that was NOT reported is outcome switching – the analysis was run and its result dropped;
- a reported statistic that was NOT declared is an addition, and twenty of them is why a nominal p of 0.05 means nothing.

Neither is misconduct on its own and both are invisible without a declaration made in advance.

Value

`prereg_declare()` a list of class `bricklayer_prereg`; `prereg_check()` a list with `ok`, `declared_not_reported`, `reported_not_declared` and a `hypotheses` data frame.

See Also

capsule_attest() to seal a declaration, capsule_falsify() for the controls themselves, falsify_family() for the correction that additions make necessary.

Examples

```
plan <- prereg_declare(c(
  ate = "use of force is higher in the exposed division",
  n_rows = "the extract has the row count the source publishes"),
  note = "OTIS 2019-2024, division-level, pre-specified")

# afterwards, against what was reported
prereg_check(plan, c("ate", "n_rows"))$ok

# dropping a declared outcome is outcome switching
prereg_check(plan, "n_rows")$declared_not_reported

# and adding undeclared ones is what makes a nominal p meaningless
prereg_check(plan, c("ate", "n_rows", "ate_by_year",
                     "ate_by_precinct"))$reported_not_declared
```

---

print.bricklayer_attestation

Printed reports for bricklayer objects

Description

Human-readable renderings of the objects the package returns. Each leads with the verdict, then the evidence. format() returns the lines as a character vector so they can be logged or written to a file; print() sends them to the console and returns its argument invisibly.

Usage

```
## S3 method for class 'bricklayer_attestation'
print(x, ...)

## S3 method for class 'bricklayer_attestation_check'
print(x, ...)

## S3 method for class 'bricklayer_bundle'
print(x, ...)

## S3 method for class 'bricklayer_bundle_check'
print(x, ...)

## S3 method for class 'bricklayer_chain'
print(x, ...)
```

```
## S3 method for class 'bricklayer_falsification'
print(x, ...)

## S3 method for class 'bricklayer_kem_key'
print(x, ...)

## S3 method for class 'bricklayer_kem_public_key'
print(x, ...)

## S3 method for class 'bricklayer_kem_capsule'
print(x, ...)

## S3 method for class 'bricklayer_mcar'
print(x, ...)

## S3 method for class 'bricklayer_power'
print(x, ...)

## S3 method for class 'bricklayer_prereg'
print(x, ...)

## S3 method for class 'bricklayer_prereg_check'
print(x, ...)

## S3 method for class 'bricklayer_falsify_family'
print(x, ...)

## S3 method for class 'bricklayer_drift'
print(x, ...)

## S3 method for class 'bricklayer_drift'
summary(object, ...)

## S3 method for class 'bricklayer_benford'
print(x, ...)

## S3 method for class 'bricklayer_signing_key'
print(x, ...)

## S3 method for class 'bricklayer_public_key'
print(x, ...)

## S3 method for class 'bricklayer_signature'
print(x, ...)

## S3 method for class 'bricklayer_benford'
summary(object, ...)
```

```
## S3 method for class 'bricklayer_env_diff'
print(x, ...)

## S3 method for class 'bricklayer_report'
print(x, ...)

## S3 method for class 'bricklayer_report'
summary(object, ...)

## S3 method for class 'bricklayer_recompute'
print(x, ...)

## S3 method for class 'bricklayer_schema'
print(x, ...)

## S3 method for class 'bricklayer_oqs_key'
print(x, ...)

## S3 method for class 'bricklayer_oqs_public_key'
print(x, ...)

## S3 method for class 'bricklayer_timestamp'
print(x, ...)

## S3 method for class 'bricklayer_certificate'
print(x, ...)

## S3 method for class 'bricklayer_certpath_check'
print(x, ...)
```

Arguments

x	The object to render.
...	Ignored, present for S3 consistency.
object	The object to summarise.

Details

Box-drawing characters are used only when the session's encoding can render them; otherwise the same layout is drawn in plain ASCII. Set the environment variable RMBL_ASCII_ONLY to force the ASCII form, which is what to do when capturing output into a fixed-width log.

Value

format() methods return a character vector; print() methods return x invisibly.

Examples

```
set.seed(7)
```

```
ref <- data.frame(v = stats::rnorm(200),
                  g = sample(c("a", "b"), 200, TRUE))
cur <- data.frame(v = stats::rnorm(200, mean = 1),
                  g = sample(c("a", "b"), 200, TRUE))

# The drift report leads with its verdict.
capsule_drift(ref, cur)

# format() gives the same lines for a log file.
head(format(capsule_drift(ref, cur)), 3)

# A Benford screen.
benford_test(10^stats::runif(500, 0, 5))

# Keys and signatures print without ever showing the secret seed.
key <- pqc_keygen(height = 2)
key
capsule_sign("a-manifest", key)
```

`profile_columns` *Column report for a data frame*

Description

One row per column: type, missingness, and the summaries that suit the column's type – mean and standard deviation alongside their robust counterparts (median, MAD) for a numeric column, and the number of distinct levels plus the most common one for a categorical column.

Usage

```
profile_columns(data, quantiles = c(0.25, 0.5, 0.75))
```

Arguments

`data`	A data frame.
`quantiles`	Quantile probabilities to include for numeric columns (default the quartiles).

Details

Reporting the classical and robust centres side by side is the point: where they disagree, the column has outliers or a heavy tail, and the mean is not describing it. That is visible in one glance here and in no single number.

Value

A data frame of class `bricklayer_profile`, one row per column: the type, the missing count and share, the number of distinct values, and for a numeric column the counts of zero, negative and infinite values, the classical and robust centre and spread, the range, the skewness, the count outside the Tukey fences, and an `inline_hist()` sketch of its distribution. A categorical column reports its most common value in `top` instead.

See Also

capsule_drift() to compare two of these, core_moments() for the underlying kernel.

Examples

```
set.seed(9)
df <- data.frame(
  clean = stats::rnorm(100),
  skewed = c(stats::rnorm(99), 500),
  grade = sample(c("a", "b", "c"), 100, TRUE),
  gappy = c(rep(NA, 10), stats::runif(90))
)

profile_columns(df)

# The mean and the median agree on `clean` and disagree sharply on
# `skewed`, which is the outlier announcing itself.
p <- profile_columns(df)
p[p$column %in% c("clean", "skewed"), c("column", "mean", "median")]

# The histogram column shows shape no summary number carries.
p[, c("column", "hist")]

# Zeros, negatives and infinities are counted separately, because
# each breaks a different downstream computation (a log, a square
# root, an average).
p[, c("column", "n_zero", "n_negative", "n_infinite")]
```

---

random_bytes *Cryptographically strong random bytes*

Description

Reads the operating system's own random source – /dev/urandom on Unix and macOS, RtlGenRandom on Windows – rather than R's Mersenne Twister.

Usage

```
random_bytes(n)
```

Arguments

n	Number of bytes (1 to 1048576).

Details

This distinction matters for anything that becomes a key. `set.seed()` makes R's generator reproducible BY DESIGN, and its state can be recovered from its output; a key drawn from it is guessable. Reading the OS source also leaves R's own random stream untouched, so generating a key does not perturb a reproducible analysis.

If no OS source can be read the function FAILS rather than falling back to a weaker generator, because a silent downgrade in a key is worse than an error.

Value

A raw vector of length n.

See Also

`derive_key()` to stretch a passphrase instead, `pqc_keygen()` which uses this for its seeds.

Examples

```
random_bytes(8)

# Independent between calls, unlike a seeded generator.
identical(random_bytes(16), random_bytes(16))

# R's own stream is not consumed, so a seeded analysis is unaffected.
set.seed(1)
a <- stats::runif(1)
set.seed(1)
invisible(random_bytes(32))
identical(stats::runif(1), a)

# As hex, for a seed argument.
paste(format(random_bytes(4)), collapse = "")
```

---

rate *Event rates per unit of population*

Description

Counts divided by exposure and scaled to a denominator, with the exact Poisson interval that says how much of the result is signal.

Usage

```
rate(x, ...)

## S3 method for class 'data.frame'
rate(
  x,
```

```
  count,
  population,
  by = NULL,
  per = 1000,
  conf_level = 0.95,
  min_count = 0,
  ...
)

## Default S3 method:
rate(x, population, per = 1000, conf_level = 0.95, min_count = 0, ...)
```

Arguments

`x`	A data frame, or a numeric vector of counts.
`...`	Passed to methods.
`count`	Column of counts: non-negative whole numbers.
`population`	Column of exposure: positive. Person-years, stops, residents – whatever the events were at risk of happening to.
`by`	Character vector of grouping columns. The rate is computed within each group.
`per`	The rate denominator: a positive number, or `"1k"`, `"10k"`, `"100k"`, `"1m"`. Default 1000.
`conf_level`	Confidence level for the interval.
`min_count`	Counts at or below this are flagged as too small to report, with the rate still computed. Default 0, which flags nothing; published guidance often uses 5 or 10.

Details

A count is not comparable across areas of different size or years of different population; a rate is. What a rate does not do is become reliable just because it is a rate: three events in a small area gives a rate with an interval several times its own width, and the interval here is the one that says so. It is the exact Poisson interval, computed through the gamma relation, so a count of zero has a lower limit of exactly zero rather than a negative number.

`per` takes a number, or one of `"1k"`, `"10k"`, `"100k"`, `"1m"`, because a denominator mistyped by a factor of ten is invisible once it reaches a table.

Value

A data frame of class `rmbl_rate` with the grouping columns, `count`, `population`, `rate`, `lower`, `upper` and `flag`, plus a `rate` attribute recording the denominator and the confidence level.

See Also

`share()` for a percentage of a total, `rate_change()` for the change in a rate between periods, `sir()` for a rate compared against an expected count rather than a population.

Examples

```
stops <- data.frame(
  division = c("North", "South", "East"),
  stops = c(412, 77, 3),
  residents = c(120000, 41000, 9500))

# per 1,000 residents, the default
rate(stops, stops, residents, by = "division")

# the denominator published guidance usually asks for
rate(stops, stops, residents, by = "division", per = "100k")

# East's three events: the interval is wider than the estimate, and
# flagging it is the point of min_count
rate(stops, stops, residents, by = "division", per = "100k",
     min_count = 5)

# vectors work too, for a single figure
rate(3, 9500, per = "100k")
```

`rate_change` *Change in a rate between periods*

Description

The change in a rate from one period to the period `lag` places earlier, with the exact conditional interval for the rate ratio.

Usage

```
rate_change(x, ...)

## S3 method for class 'data.frame'
rate_change(
  x,
  count,
  population,
  period,
  by = NULL,
  lag = 1L,
  per = 1000,
  conf_level = 0.95,
  min_count = 0,
  ...
)
```

Arguments

x	A data frame.
...	Passed to methods.
count	Column of counts: non-negative whole numbers.
population	Column of exposure: positive.
period	Column of periods. Sorted, and compared on the period value rather than on row position, so a missing year gives no comparison instead of a silent comparison against the wrong year.
by	Character vector of grouping columns.
lag	How many periods back to compare against. Default 1.
per	The rate denominator: a positive number, or "1k", "10k", "100k", "1m".
conf_level	Confidence level for the interval.
min_count	Comparisons where the earlier count is at or below this are flagged and their percent change withheld, the way yoy() withholds a percent change off a tiny base.

Details

This is not the percent change of two rates treated as measured numbers. Both the counts and the denominators move between periods, and an interval that ignores the denominators understates the uncertainty of the change. The construction here conditions on the total of the two counts and corrects for the ratio of the two exposures, which is yoy()'s exact conditional-binomial interval generalised to unequal denominators: with equal populations it reduces to exactly that.

Value

A data frame of class rmbl_rate_change with the grouping columns, period, count, population, rate, previous_rate, rate_ratio, pct_change, pct_lower, pct_upper and flag.

See Also

rate(), yoy() for change in a count or a measured quantity.

Examples

```
d <- data.frame(
  year = rep(2021:2023, each = 2),
  division = rep(c("North", "South"), 3),
  stops = c(400, 70, 430, 66, 455, 61),
  residents = c(120000, 41000, 122000, 41500, 125000, 42000))

# North's count rose while its population rose too: the rate change
# is smaller than the count change, which is the reason to use it
rate_change(d, stops, residents, year, by = "division", per = "100k")
```

record *Record a Cross-Check Result in a Manifest*

Description

Appends one named cross-check entry to a manifest, classifying it as PASS, DIFFER, or INFO, printing a formatted line to the console, and returning the updated manifest.

Usage

```
record(
  manifest,
  name,
  observed,
  expected,
  tol = 1e-04,
  group = "general",
  synthetic = FALSE
)
```

Arguments

manifest	A manifest as returned by make_manifest().
name	Unique name for this cross-check; used as the result key.
observed	The observed value (numeric or otherwise).
expected	The expected value to compare against.
tol	Numeric tolerance; a numeric pair within tol is PASS. Defaults to 0.0001.
group	Optional grouping label for the entry. Defaults to "general".
synthetic	Logical; if TRUE the entry is marked INFO because comparison against synthetic data is not meaningful.

Value

The updated manifest, returned so calls can be chained.

Examples

```
man <- make_manifest(list(project = "demo"), environment = FALSE)

# Within tolerance -> PASS.
man <- record(man, "mean_matches", observed = 1.0001, expected = 1,
              tol = 0.001)
man$results$mean_matches$status      # "PASS"

# Outside tolerance -> DIFFER.
man <- record(man, "sd_matches", observed = 2.5, expected = 2.0, tol = 0.01)
```

```
man$results$sd_matches$status        # "DIFFER"

# Synthetic data -> INFO (comparison not meaningful).
man <- record(man, "synthetic_row", observed = 5, expected = 5,
              synthetic = TRUE)
man$results$synthetic_row$status     # "INFO"

# Calls chain: record() returns the mutated manifest.
length(man$results)                  # 3
```

`region_coverage` *Population of the regions that contain a unit, and of those that do not*

Description

Summarises where a set of point-located units sits relative to the regions of a statistical geography: how many regions hold at least one unit, and what share of the population lives in them.

Usage

```
region_coverage(region, population, units)
```

Arguments

`region`	Region identifiers, one per region, not repeated.
`population`	Population of each region, same length and order.
`units`	Number of units located in each region. Zero is the expected value for most regions in most geographies.

Details

The share this returns is CONTEXT, not a denominator, and the distinction is the reason the function exists rather than a bare `tapply()`.

A region holding no unit is not an unserved population. Units serve catchments, and a catchment is an administrative fact about where people are sent from; a point location does not state it and cannot imply it. Some geographies were never meant to have one unit each.

So a rate built by summing the populations of unit-holding regions pairs a denominator covering part of the territory with a numerator drawn from all of it. Every such rate is inflated, and inflated unevenly: a dense region holding one unit and a sparse region holding seven distort it in opposite directions. Where numerator and denominator must cover the same population, the defensible figure is the whole-territory one.

The print method says this each time, because the covered share is precisely the number a reader is tempted to divide by.

Value

A data frame of class `rmbl_region_coverage`, one row per region, with the columns `region`, `population`, `units`, `has_unit` and `pop_share`, ordered by units then population. The totals are carried on the `coverage` attribute and printed by the print method.

See Also

`region_map_integrity()`, `region_map_compare()`, `region_map_second_route()`

Examples

```
# Four regions, two of which hold a facility.
cov <- region_coverage(region = c("A", "B", "C", "D"),
                       population = c(1200000, 800000, 450000, 90000),
                       units = c(3, 0, 1, 0))
cov

# The covered share is reported, and is not a rate denominator.
attr(cov, "coverage")$covered_share
```

---

region_map_compare *Compare a recomputed region map against a published one*

Description

Matches two region maps on the unit identifier and compares the named columns cell by cell.

Usage

```
region_map_compare(published, observed, unit, cols = NULL)
```

Arguments

`published`	The region map as published.
`observed`	The region map as recomputed.
`unit`	Name of the unit identifier column, present in both.
`cols`	Columns to compare. Defaults to every column the two share apart from `unit`.

Details

Numeric columns are compared with `all.equal()` at its default tolerance, so a coordinate that survived a round trip through text is not reported as a change; everything else is compared exactly after trimming whitespace.

What this establishes is that the recomputation reproduced the published assignment. It does NOT establish that the assignment is right: run the same method against the same boundary file and a definitional error reproduces perfectly. That is what `region_map_second_route()` is for.

Value

A data frame, one row per compared column plus a `rows` row for units present on one side only: `column`, `cells`, `mismatched` and `first`, the first disagreement written out as text. Zero `mismatched` throughout is the passing result.

See Also

`region_map_second_route()`, `region_map_integrity()`

Examples

```
pub <- data.frame(inst = c("North Jail", "South Jail"),
                  cd = c("3557", "3520"), stringsAsFactors = FALSE)
obs <- pub
region_map_compare(pub, obs, "inst")

# a changed assignment is reported with the unit that moved
obs$cd[2] <- "3521"
region_map_compare(pub, obs, "inst")
```

---

region_map_from_points

Recompute a region map by point in polygon

Description

Assigns each point to the polygon that contains it. Requires the `sf` package and a boundary file; returns `NULL` when either is absent, so a verification script can record the check as unavailable instead of failing for a missing optional dependency.

Usage

```
region_map_from_points(x, y, unit, boundaries, fields, crs = 4326)
```

Arguments

`x`, `y`	Longitude and latitude of each point.
`unit`	Identifier for each point, same length.
`boundaries`	Path to a boundary file `sf` can read.
`fields`	Columns of the boundary file to carry onto the result.
`crs`	Coordinate reference system the points are in. Default `4326`, that is WGS84, which is what published latitude and longitude columns almost always are.

Details

The points are projected onto the boundary file's own coordinate reference system before matching, never the other way round: a cartographic boundary file is published in a projection chosen for the country it covers, and reprojecting the polygons to compare them against unprojected points moves the edges.

Point in polygon is preferred to matching place names against subdivision names whenever both are available. Names fail on exactly the cases that matter and fail quietly: a facility in a community rather than an incorporated municipality, a city amalgamated into a larger one, a township absorbed by a neighbour. The geometry has no opinion about any of that.

Use `region_map_second_route()` to check this result against the name route, with those cases named.

Value

A data frame: `unit`, the requested `fields`, and `n_regions`, the number of polygons that contained the point. Any value of `n_regions` other than 1 is a failure – zero means the point fell outside the geography, more than one means the boundaries overlap – so it is returned rather than silently resolved. `NULL` if `sf` is not installed or `boundaries` does not exist.

See Also

`region_map_compare()`, `region_map_second_route()`

Examples

```
# Needs sf and a boundary file, so this is the shape of the call
# rather than a run of it.
## Not run:
obs <- region_map_from_points(
  x = inst$Longitude, y = inst$Latitude, unit = inst$Institution,
  boundaries = "lcd_000b21a_e.shp", fields = c("CDUID", "CDNAME"))
stopifnot(all(obs$n_regions == 1))

## End(Not run)
```

---

region_map_integrity *Internal soundness of a region map*

Description

Checks that a region map assigns exactly one region to every unit and that every region it names is one the reference geography knows about.

Usage

```
region_map_integrity(map, unit, region, regions = NULL)
```

Arguments

map	Data frame, one row per unit.
unit	Name of the column holding the unit identifier.
region	Name of the column holding the region identifier.
regions	Optional character vector of every valid region identifier, typically the identifier column of the population table. When supplied, region codes outside it are counted as failures.

Details

These are the failures that a comparison against published output cannot see, because they would be present on both sides: a unit matched into two regions, a unit matched into none, a region code that is a typo or belongs to a neighbouring province. None of them require the geometry, so they run with nothing installed.

Value

A data frame with one row per check: `check`, `observed`, `expected` and `pass`. Every check is stated so that zero is the passing value, which is what makes the frame safe to feed straight into a manifest.

See Also

`region_map_compare()`, `region_map_second_route()`, `region_coverage()`

Examples

```
cw <- data.frame(inst = c("North Jail", "South Jail", "East Jail"),
                 cd = c("3557", "3520", "3506"),
                 stringsAsFactors = FALSE)
region_map_integrity(cw, "inst", "cd", regions = c("3557", "3520", "3506"))

# a region code the geography does not know fails the third check
cw$cd[3] <- "2406"
region_map_integrity(cw, "inst", "cd", regions = c("3557", "3520", "3506"))
```

---

region_map_second_route

Check a region map against an independently derived assignment

---

Description

Compares the region each unit was assigned with the region a DIFFERENT method assigns it, and marks the disagreements that are already known and explained.

Usage

```
region_map_second_route(map, unit, region, route, known = character())
```

Arguments

map	Data frame, one row per unit.
unit	Name of the unit identifier column.
region	Name of the assigned region column.
route	Named character vector, or a data frame with the same two column names, giving the second method's assignment. Units it does not cover are skipped rather than counted as disagreements.
known	Units whose disagreement is expected and documented – the cases the second route is known to get wrong.

Details

This is the only check here that can catch an error in the original method, because it does not use that method. Recomputing point in polygon against the same boundary file proves the pipeline is deterministic; deriving the region a second way – from a name, a postal geography, an administrative lookup – can disagree, and a disagreement is information either way round.

The known argument exists because a second route usually has understood weaknesses: a place name that is a community rather than a municipality, an amalgamated city, a township absorbed into a neighbour. Listing them keeps the check sharp instead of loosening the tolerance until everything passes, and listing them by NAME means an unexpected disagreement cannot hide inside an allowance.

Value

A data frame of disagreements: unit, primary, second and known. sum(!x$known) is the number of unexplained disagreements and zero is the passing value; nrow(x) should equal the number of documented cases, because a documented case that stops disagreeing means the second route changed underneath the documentation.

See Also

region_map_compare(), region_map_integrity()

Examples

```
cw <- data.frame(inst = c("North Jail", "South Jail", "Hill Jail"),
                 cd = c("3557", "3520", "3506"),
                 stringsAsFactors = FALSE)

# a name-based route that is known to mis-place one unit
route <- c("North Jail" = "3557", "South Jail" = "3520",
           "Hill Jail" = "3519")
region_map_second_route(cw, "inst", "cd", route, known = "Hill Jail")

# an undocumented disagreement is what the check is for
route["South Jail"] <- "3521"
d <- region_map_second_route(cw, "inst", "cd", route, known = "Hill Jail")
sum(!d$known)
```

`report_markdown` *Write a capsule report as Markdown*

Description

Renders a `capsule_report()` as Markdown, so the assessment can travel with the capsule instead of living in a console someone has since closed.

Usage

```
report_markdown(report, path = NULL, title = "Capsule report")
```

Arguments

`report`	A `bricklayer_report`.
`path`	Optional file to write. Without one the lines are returned.
`title`	Heading for the document.

Value

The Markdown lines, invisibly when written to a file.

See Also

`capsule_report()`

Examples

```
set.seed(1)
df <- data.frame(v = stats::rnorm(100), g = rep("x", 100),
                 stringsAsFactors = FALSE)
r <- capsule_report(df)

md <- report_markdown(r)
cat(head(md, 8), sep = "\n")

# Written beside the capsule it describes.
p <- tempfile(fileext = ".md")
report_markdown(r, p)
file.exists(p)
unlink(p)
```

resolve_via_arcgis *Resolve a Query URL via ArcGIS FeatureServer Metadata*

Description

Verifies that an ArcGIS FeatureServer layer still exists by fetching its `f=json` metadata, then returns a paged GeoJSON query URL for the full layer. ArcGIS FeatureServer layers back the Toronto Police Service open-data portal used across the MORIE family.

Usage

```
resolve_via_arcgis(provenance)
```

Arguments

provenance	A provenance list as returned by `load_provenance()`. Must contain `dataset$arcgis_layer_url`, a FeatureServer layer root such as `"https://services.arcgis.com/.../Assault_Open_Data/Featur`

Value

The layer query URL (`where=1=1`, all fields, GeoJSON) as a character string, or `NULL` if the field is missing, the request fails, or the layer metadata reports an error.

Examples

```
# Missing fields return NULL rather than erroring:
resolve_via_arcgis(list())

prov <- list(dataset = list(arcgis_layer_url = paste0(
  "https://services.arcgis.com/S9th0jAJ7bqgIRjw/arcgis/rest/services/",
  "Neighbourhood_Crime_Rates_Open_Data/FeatureServer/0")))
resolve_via_arcgis(prov)
```

resolve_via_ckan *Resolve a Download URL via CKAN package_show*

Description

Queries the CKAN `package_show` endpoint recorded in a provenance object and returns the URL of the first resource whose name matches the provenance's name-match pattern. CKAN powers data.ontario.ca, data.gov.uk, data.gov, and most government open-data portals, so this recovers the current download URL even if the underlying resource UUID has been replaced.

Usage

```
resolve_via_ckan(provenance)
```

Arguments

provenance	A provenance list as returned by load_provenance(). Must contain dataset$ckan_api_endpoint and resource$name_match_pattern.

Value

The matched resource URL as a character string, or NULL if the endpoint is missing, the request fails, CKAN reports failure, or no resource name matches.

Examples

```
# Missing fields return NULL rather than erroring:
resolve_via_ckan(list())

prov <- list(
  dataset  = list(ckan_api_endpoint = paste0(
    "https://data.ontario.ca/api/3/action/package_show",
    "?id=ontario-public-library-statistics")),
  resource = list(name_match_pattern = "2014")
)
resolve_via_ckan(prov)
```

---

resolve_via_ckan_search

Resolve a Download URL via CKAN package_search

Description

Fallback for resolve_via_ckan() when the dataset slug has changed. Derives the CKAN portal base URL from the provenance's package_show endpoint, runs a package_search query (from resource$search_query, or derived from the name-match pattern), and returns the URL of the first matching resource, preferring CSV format when specified.

Usage

```
resolve_via_ckan_search(provenance)
```

Arguments

provenance	A provenance list as returned by load_provenance(). Uses resource$search_query, resource$name_match_pattern, resource$format, and dataset$ckan_api_endpoint.

Value

The matched resource URL as a character string, or NULL if no query or base URL can be derived, the request fails, or nothing matches.

Examples

```
# Missing fields return NULL rather than erroring:
resolve_via_ckan_search(list())

prov <- list(
  dataset  = list(ckan_api_endpoint = paste0(
    "https://data.ontario.ca/api/3/action/package_show",
    "?id=ontario-public-library-statistics")),
  resource = list(name_match_pattern = "2014",
                  search_query = "public library statistics")
)
resolve_via_ckan_search(prov)
```

`resolve_via_socrata` *Resolve a Download URL via the Socrata Metadata API*

Description

Verifies that a Socrata dataset still exists by fetching its `api/views` metadata, then returns the canonical CSV export URL. Socrata powers the Calgary, Chicago, and NYC open-data portals used across the MORIE family.

Usage

```
resolve_via_socrata(provenance)
```

Arguments

`provenance`	A provenance list as returned by `load_provenance()`. Must contain `dataset$socrata_domain` (e.g. `"data.cityofchicago.org"`) and `dataset$socrata_id` (the 4x4 dataset id, e.g. `"ijzp-q8t2"`) .

Value

The CSV export URL as a character string, or `NULL` if the fields are missing, the request fails, or the metadata reports an error.

Examples

```
# Missing fields return NULL rather than erroring:
resolve_via_socrata(list())

prov <- list(dataset = list(socrata_domain = "data.cityofchicago.org",
                            socrata_id     = "ijzp-q8t2"))
resolve_via_socrata(prov)
```

`revocation_fetch` *Fetch revocation data for a certificate path*

Description

Retrieves CRLs from the distribution points named in the certificates, and asks any OCSP responder they name about each one. Called by `cert_chain_verify()` when `revocation = "fetch"`.

Usage

```
revocation_fetch(path, timeout = 10)
```

Arguments

`path`	A certificate path, leaf first, as `cert_chain_verify()` builds it.
`timeout`	Seconds to allow each request.

Details

The OCSP request is sent by GET with the DER request base64-encoded into the URL, as RFC 6960 appendix A.1.1 allows. That avoids needing to POST, and works with responders that accept it; one that requires POST is reported as unreachable rather than treated as a pass.

A responder's answer is only believed when its signature verifies under a certificate in the path or one it carries that the path issued. An unsigned or unverifiable answer is reported as such – treating it as "good" would make revocation checking worse than skipping it, since it would look like it had happened.

Value

A list with `crls` (raw vectors fetched), `ok` (a logical per note) and `notes` (a named list of details).

See Also

`cert_chain_verify()`, `timestamp_verify()`.

Examples

```
# Reaches the network, so it is not run here.
## Not run:
res <- cert_chain_verify("leaf.crt", trust = "ca.crt",
                         revocation = "fetch")
res$checks

## End(Not run)
```

`rmbl_base64` *Base64 encoding*

Description

Encodes and decodes base64, in both the standard alphabet and the URL-safe variant. Computed in R with no dependency, so a capsule can embed binary content in a text manifest wherever this package runs.

Usage

```
bricklayer_json_base64_enc(input)

bricklayer_json_base64_dec(input)

bricklayer_json_base64url_enc(input)

bricklayer_json_base64url_dec(input)
```

Arguments

`input`	For the encoders, a raw vector or a character vector (joined with newlines first). For the decoders, base64 text or its raw bytes.

Details

`bricklayer_json_base64_enc()` breaks its output into 72-character lines, matching jsonlite's encoder; the decoder ignores line breaks and any other character outside the alphabet, so either form round trips.

The URL-safe variant substitutes `-` and `_` for `+` and `/` and drops the `=` padding, which is what makes it safe in a URL path, a query string or a filename.

Base64 is an ENCODING, not encryption or a digest: it hides nothing and anyone can reverse it. Use `core_sha256()` to pin content and `core_hmac_sha256()` to authenticate it.

Value

The encoders return a length-1 character vector (`NA_character_` for `NULL` input); the decoders return a raw vector.

See Also

`json_gzip_encode()`, which composes this with gzip.

Examples

```
# Round trip through the standard alphabet.
b <- bricklayer_json_base64_enc("hello capsule")
b
rawToChar(bricklayer_json_base64_dec(b))

# Raw input works the same way.
bricklayer_json_base64_enc(charToRaw("abc"))
bricklayer_json_base64_dec(bricklayer_json_base64_enc(charToRaw("abc")))

# Padding appears when the length is not a multiple of three.
bricklayer_json_base64_enc("a")
bricklayer_json_base64_enc("ab")
bricklayer_json_base64_enc("abc")

# The URL-safe variant has no "+", "/" or "=" to escape.
bricklayer_json_base64url_enc(as.raw(c(255, 224, 63)))
bricklayer_json_base64_enc(as.raw(c(255, 224, 63)))
bricklayer_json_base64url_dec(
  bricklayer_json_base64url_enc("path/safe?yes")
)

# Long input is wrapped, and the decoder ignores the breaks.
long <- bricklayer_json_base64_enc(strrep("x", 200))
grepl("\n", long)
rawToChar(bricklayer_json_base64_dec(long)) == strrep("x", 200)
```

`rmbl_chain` *Tamper-evident chain of capsule manifests*

Description

Each entry records the digest of the entry before it, so the chain's integrity covers the ORDER and COMPLETENESS of the history, not merely the contents of each manifest. Deleting an entry, inserting one, or editing one breaks the links from that point onward, and `chain_verify()` reports the first index where the break occurs.

Usage

```
chain_new()

chain_append(chain, entry, label = NA_character_)

chain_head(chain)

chain_seal(chain)

chain_verify(chain)
```

Arguments

chain	A chain from chain_new() or chain_append().
entry	Any object to record. It is digested through its own deterministic serialization, so lists and data frames are accepted as readily as strings.
label	Optional short character label for the entry.

Details

This is the structure behind an append-only audit log, and it is what a per-manifest digest alone cannot give you: individually valid manifests say nothing about whether any were removed.

Value

chain_new() and chain_append() return an object of class bricklayer_chain. chain_verify() returns a list with valid, n, broken_at (NA when intact) and head. chain_head() returns the head digest. chain_seal() returns a single digest over the verified chain's length and links, or NA if the chain does not verify.

What the head covers, and what it does not

chain_head() is the STORED digest of the last entry. Deleting an entry from the MIDDLE leaves that value untouched – the stored digests do not change, only the links between them stop agreeing – so the head alone will not notice. chain_verify() will, and names the index.

Conversely, truncating from the END leaves a perfectly valid prefix that chain_verify() accepts, while the head changes.

The two failures are complementary, which is why chain_seal() exists: it verifies the links AND folds the entry count and every link digest into one value, so a single signature over the seal detects an edit, a deletion, a reordering, an insertion and a truncation alike. Sign the seal, not the head.

Signing is what turns tamper-EVIDENT into tamper-PROOF: without a signature an attacker who rewrites the whole chain leaves it internally consistent, because recomputing every link is cheap.

See Also

capsule_sign() to sign the head, merkle_root() for pinning the contents of one capsule rather than a history.

Examples

```
ch <- chain_new()
ch <- chain_append(ch, "manifest for run 1", label = "run-1")
ch <- chain_append(ch, "manifest for run 2", label = "run-2")
ch <- chain_append(ch, "manifest for run 3", label = "run-3")
ch

# An intact chain verifies.
chain_verify(ch)$valid

# Editing an entry breaks it, and names where.
```

```
edited <- ch
edited$entries[[2]]$digest <- core_sha256("something else")
chain_verify(edited)$valid
chain_verify(edited)$broken_at

# So does deleting one, which a per-manifest digest would not catch.
dropped <- ch
dropped$entries[[2]] <- NULL
chain_verify(dropped)$valid

# Sign the SEAL, which covers the links and the length together.
key <- pqc_keygen(height = 2)
sig <- capsule_sign(chain_seal(ch), key)
capsule_verify(chain_seal(ch), sig, signing_public_key(key))

# Truncating the chain still seals, but to a different value, so the
# signature no longer verifies.
truncated <- ch
truncated$entries[[3]] <- NULL
capsule_verify(chain_seal(truncated), sig, signing_public_key(key))

# A chain whose links disagree has no seal to present at all.
chain_seal(dropped)
```

`rmbl_core_rank` *Rank correlation and midranks (C backend)*

Description

`core_cor_spearman()` is Spearman's rho: the Pearson correlation of the ranks, so it measures monotone association rather than linear association and is unaffected by any order-preserving transformation of either variable. `core_midranks()` exposes the ranks themselves; tied values share the average of the ranks they span, which is what makes the result agree with `stats::cor()` on tied data.

Usage

```
core_cor_spearman(x, y)

core_midranks(x)
```

Arguments

`x`, `y`	Numeric vectors of the same length.

Value

`core_cor_spearman()` a length-1 numeric in \ [-1, 1]; `core_midranks()` a numeric vector the length of `x`.

[-1, 1]: R:-1,%201%5C

Examples

```
x <- c(1, 2, 3, 4, 5)
y <- c(2, 4, 9, 16, 25)

# Perfectly monotone but not linear: rho is 1 where Pearson is not.
core_cor_spearman(x, y)
core_cor(x, y)

# Agrees with stats::cor(), ties included.
xt <- c(1, 2, 2, 2, 5, 5, 7)
yt <- c(3, 1, 1, 4, 4, 9, 2)
all.equal(core_cor_spearman(xt, yt), stats::cor(xt, yt, method = "spearman"))

# Tied values share the average of the ranks they cover.
core_midranks(xt)
all.equal(core_midranks(xt), rank(xt))

# Invariant to any monotone rescaling.
all.equal(core_cor_spearman(x, y), core_cor_spearman(exp(x), log(y)))
```

---

rmbl_core_robust *Quantiles, median and robust spread (C backend)*

Description

`core_quantile()` is the type-7 quantile, which is R's default, so it agrees with `stats::quantile(x, probs, type = 7)`. `core_median()` is the 50% point. `core_mad()` is the median absolute deviation, scaled by `constant` so that it estimates the standard deviation of a normal sample. `core_iqr()` is the interquartile range and `core_tukey_fences()` the outlier fences drawn at `k` IQRs beyond the quartiles.

Usage

```
core_quantile(x, probs = c(0, 0.25, 0.5, 0.75, 1))

core_median(x)

core_mad(x, constant = 1.4826)

core_iqr(x)

core_tukey_fences(x, k = 1.5)
```

Arguments

`x`	Numeric vector (coerced with `as.numeric()`) .
`probs`	Numeric vector of probabilities in \ [0, 1]. [0, 1]: R:0,%201%5C

constant — Scale factor for core_mad(). The default 1.4826 makes the MAD consistent for the standard deviation under normality, and is the same rounded value stats::mad() uses, so the two agree exactly. The unrounded consistency constant is 1 / qnorm(3/4) = 1.4826022185...; pass it explicitly if you want the extra digits, or 1 for the unscaled median deviation.

k — Fence width in IQRs (default 1.5, Tukey's convention; 3 is the usual "far out" cutoff).

Details

These are the robust counterparts of core_moments(): a single corrupted row can move a mean or a variance arbitrarily far, but moves a median or a MAD hardly at all – which is what you want when deciding whether a freshly fetched column is still the column a capsule was pinned against.

Value

core_quantile() returns a numeric vector the length of probs; core_median(), core_mad() and core_iqr() a length-1 numeric; core_tukey_fences() a named length-2 numeric (lower, upper) .

Examples

```
x <- c(2, 4, 4, 4, 5, 5, 7, 9)
core_quantile(x, c(0.25, 0.5, 0.75))
all.equal(core_quantile(x, c(0.1, 0.9)),
          as.numeric(stats::quantile(x, c(0.1, 0.9))))

core_median(x)
core_mad(x)
all.equal(core_mad(x), stats::mad(x))
core_mad(x, constant = 1)                  # unscaled median deviation
core_mad(x, constant = 1 / stats::qnorm(3/4))  # unrounded constant

core_iqr(x)
core_tukey_fences(x)

# Robustness: one wild value barely moves the median, but moves the
# mean a long way.
wild <- c(x, 1000)
c(mean = mean(wild), median = core_median(wild))

# Values outside the fences are the candidates to inspect.
f <- core_tukey_fences(wild)
wild[wild < f["lower"]] | wild > f["upper"]]
```

`rmbl_core_spread` *Standard deviation and Euclidean distance (C backend)*

Description

`core_sd()` is the square root of the variance computed by the shared core; `core_dist()` is the Euclidean distance between two equal-length vectors.

Usage

```
core_sd(x, ddof = 1L)

core_dist(a, b)
```

Arguments

x, a, b	Numeric vectors (coerced with `as.numeric()`) .
ddof	Denominator degrees of freedom. The default `1` gives the sample standard deviation, matching `stats::sd()`; `0` gives the population figure.

Details

NA/NaN propagate – there is no `na.rm`. Call `stats::na.omit()` first if you need NA handling.

Value

A length-1 numeric.

See Also

`core_moments()` for the mean, variance, skewness and kurtosis in a single pass.

Examples

```
# Sample standard deviation, matching stats::sd().
core_sd(c(2, 4, 4, 4, 5, 5, 7, 9))
all.equal(core_sd(1:10), stats::sd(1:10))

# ddof = 0 divides by n instead of n - 1.
core_sd(1:10, ddof = 0)
all.equal(core_sd(1:10, ddof = 0), sqrt(mean((1:10 - mean(1:10))^2)))

# Euclidean distance between two points.
core_dist(c(0, 0), c(3, 4))     # 5
core_dist(1:5, 1:5)             # 0 -- a point is zero from itself
```

`rmbl_core_stats` *Fast summary statistics (C backend)*

Description

Thin R wrappers over the `rmoriebricklayer` compiled core – the same kernels that sibling packages reach through `LinkingTo: rmoriebricklayer`. NA/NaN values propagate (there is no `na.rm`); call `stats::na.omit()` first if you need NA handling.

Usage

```
core_mean(x)

core_var(x)

core_cor(x, y)
```

Arguments

`x`, `y`	Numeric vectors (coerced with `as.numeric()`).

Value

`core_mean()`, `core_var()` and `core_cor()` return a length-1 numeric. `core_var()` uses the `n - 1` (sample) denominator, matching `stats::var()`.

Examples

```
## core_mean(): sample mean (NA/NaN propagate; no na.rm)
core_mean(1:10)                 # 5.5
core_mean(c(2.5, 3.5))          # 3
core_mean(c(1, 2, NA))          # NA -- call stats::na.omit() first if needed
core_mean(stats::na.omit(c(1, 2, NA)))

## core_var(): n-1 (sample) variance, matching stats::var()
core_var(c(2, 4, 4, 4, 5, 5, 7, 9))
all.equal(core_var(1:10), stats::var(1:10))   # agrees with base R

## core_cor(): Pearson correlation of two equal-length vectors
core_cor(1:10, (1:10)^2)        # strong positive, near 0.97
core_cor(1:10, 10:1)            # perfect negative: -1
```

`rmbl_core_trimmed` *Trimmed and winsorized means (C backend)*

Description

Two ways to stop a handful of extreme rows dominating a column's centre. `core_trimmed_mean()` DISCARDS the `floor(n * trim)` largest and smallest values, matching `mean(x, trim = )`. `core_winsorized_mean()` instead PULLS THEM IN to the most extreme surviving values, so every observation still contributes weight – usually the better choice when the extremes are real measurements rather than errors.

Usage

```
core_trimmed_mean(x, trim = 0.1)

core_winsorized_mean(x, trim = 0.1)
```

Arguments

`x`	Numeric vector (coerced with `as.numeric()`) .
`trim`	Proportion trimmed from *each* end, in \ [0, 0.5]. At `0.5` both reduce to the median. [0, 0.5]: R:0,%200.5%5C

Value

A length-1 numeric.

Examples

```
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 100)

mean(x)                              # dragged up by the 100
core_trimmed_mean(x, 0.1)            # the 100 and the 1 dropped
core_winsorized_mean(x, 0.1)         # the 100 pulled back to 9

# Agrees with base R's own trimming.
all.equal(core_trimmed_mean(x, 0.2), mean(x, trim = 0.2))

# trim = 0 is the plain mean; trim = 0.5 is the median.
all.equal(core_trimmed_mean(x, 0), mean(x))
all.equal(core_trimmed_mean(x, 0.5), core_median(x))
```

`rmbl_distinct` *Distinct-value count in fixed memory*

Description

HyperLogLog: estimates how many distinct values a column holds using a fixed 4-byte register per bucket – 64 KB at the default `p = 14` – regardless of the column's length or cardinality. Exact counting needs memory proportional to the number of distinct values, which is the thing you cannot afford on a capsule member of unknown size.

Usage

```
distinct_sketch(x, p = 14L, registers = NULL)

distinct_count(registers)

sketch_merge(a, b)
```

Arguments

`x`	A vector; coerced to character, since distinctness is compared on the rendered value. `NA` is skipped.
`p`	Log2 of the register count, 4 to 20 (default 14).
`registers`	Registers from a previous call, to fold another chunk into the same sketch.
`a, b`	Register sets to merge.

Details

The estimate carries a relative standard error of about `1.04 / sqrt(2^p)`, so 0.8% at the default. It is an ESTIMATE: use `length(unique(x))` when the column fits in memory and an exact answer matters. Below roughly `2.5 * 2^p` distinct values the estimator switches to linear counting, which is near-exact in that range.

Values are hashed with the package's SHA-256, so the sketch is identical on every platform and across sessions.

Value

`distinct_sketch()` an integer vector of registers; `distinct_count()` a length-1 numeric estimate; `sketch_merge()` the element-wise maximum of two register sets.

References

Flajolet P, Fusy E, Gandouet O, Meunier F (2007). HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Analysis of Algorithms 2007*, 137–156.

Examples

```
set.seed(1)
x <- sample(1:5000, 200000, replace = TRUE)

# Close to the true 5000 distinct values, in fixed memory.
distinct_count(distinct_sketch(x))
length(unique(x))

# Small cardinalities are near-exact, via linear counting.
distinct_count(distinct_sketch(c("a", "b", "c", "a", "b")))

# Chunks fold into one sketch, so a file can be counted block by
# block, and two independent sketches can be merged.
s <- distinct_sketch(x[1:100000])
s <- distinct_sketch(x[100001:200000], registers = s)
distinct_count(s)

a <- distinct_sketch(x[1:100000])
b <- distinct_sketch(x[100001:200000])
distinct_count(sketch_merge(a, b))

# An empty input has no distinct values.
distinct_count(distinct_sketch(character(0)))
```

---

rmbl_drop *Drop empty or constant columns and rows*

Description

`drop_empty()` removes rows or columns that are entirely missing. `drop_constant()` removes columns that hold a single distinct value. The counterparts of `janitor::remove_empty()` and `janitor::remove_constant()`.

Usage

```
drop_empty(data, which = c("rows", "cols"))

drop_constant(data, na_as_value = FALSE)
```

Arguments

`data`	A data frame.
`which`	`"rows"`, `"cols"`, or both (the default).
`na_as_value`	Treat `NA` as a distinct value, so a column of `NA` s plus one real value counts as two (default `FALSE`) .

Details

A constant column carries no information and breaks anything that scales by variance, so it is worth removing – but it is also a FINDING. A column that was informative in the pinned capsule and is constant in a fresh fetch means the source changed, so check `capsule_drift()` before deleting it and moving on.

Value

The data frame, with the offending rows or columns removed. The names of what was dropped are attached as the `"dropped"` attribute.

See Also

`profile_columns()`, which reports `n_distinct` without removing anything.

Examples

```
df <- data.frame(
  keep = c(1, 2, NA),
  all_na = c(NA, NA, NA),
  constant = c(7, 7, 7),
  stringsAsFactors = FALSE
)

drop_empty(df)
attr(drop_empty(df), "dropped")

drop_constant(df)

# Rows only.
drop_empty(data.frame(a = c(1, NA), b = c(2, NA)), which = "rows")

# A column of NAs plus one value is constant by default, and not when
# NA is treated as a value of its own.
x <- data.frame(v = c(NA, NA, 5))
ncol(drop_constant(x))
ncol(drop_constant(x, na_as_value = TRUE))
```

`rmbl_file_digest` *SHA-512 and CRC-32 of a file*

Description

Streams the file in blocks, so memory use does not grow with the file. The SHA-256 counterpart is `sha256_file()`.

Usage

```
sha512_file(path, block_bytes = 1048576L)

crc32_file(path, block_bytes = 1048576L)
```

Arguments

path	Path to an existing file.
block_bytes	Read size in bytes (default 1048576). Affects speed only, never the result.

Value

A length-1 character vector (sha512_file()) or numeric (crc32_file()) .

Examples

```
p <- tempfile()
writeLines("capsule payload", p)

sha512_file(p)
crc32_file(p)

# The block size is a speed knob and cannot change the digest.
identical(sha512_file(p, 16), sha512_file(p, 1048576))

unlink(p)
```

---

rmbl_json_gzip *Compressed, base64-encoded JSON*

Description

json_gzip_encode() serialises to JSON, compresses with gzip and encodes the result as base64, so it can be embedded in another JSON document, a header, or a text column. json_gzip_decode() reverses all three steps.

Usage

```
json_gzip_encode(x, raw = FALSE, ...)

json_gzip_decode(txt, raw = FALSE, ...)
```

Arguments

x	Object to encode.
raw	Return (or accept) gzip bytes instead of base64.
...	Passed to bricklayer_json_to_json().
txt	Base64 string (or raw vector) from json_gzip_encode().

Details

JSON is highly compressible because every record repeats the key names, so this is usually a large saving on anything record-shaped – but it is opaque. Use it for payloads that travel, and plain `bricklayer_json_to_json()` for anything a person is meant to read or a `git diff` is meant to show.

`raw = TRUE` skips the base64 step and returns the gzip bytes, which is what to use when writing to a file rather than embedding in text.

Value

`json_gzip_encode()` a length-1 character vector, or a raw vector when `raw = TRUE`. `json_gzip_decode()` the decoded object.

See Also

`bricklayer_json_to_json()`, `bricklayer_json_serialize()` for a lossless but uncompressed form.

Examples

```
x <- list(rows = data.frame(id = 1:50, value = stats::runif(50)))

enc <- json_gzip_encode(x)
substring(enc, 1, 40)

# Smaller than the JSON it came from, because the keys repeat.
c(json = nchar(bricklayer_json_to_json(x)), gzip_b64 = nchar(enc))

# Round trips.
identical(json_gzip_decode(enc)$rows$id, 1:50)

# Raw gzip bytes, for writing to a file.
bytes <- json_gzip_encode(x, raw = TRUE)
class(bytes)
identical(json_gzip_decode(bytes, raw = TRUE)$rows$id, 1:50)
```

`rmbl_json_serialize` *Lossless JSON serialisation of an R object*

Description

Writes an R object to JSON with its type and attributes alongside the value, so the round trip returns THE SAME OBJECT rather than something that merely prints the same. The counterpart of jsonlite's `serializeJSON()` / `unserializeJSON()`, computed by this package's own codec with no jsonlite dependency.

Usage

```
bricklayer_json_serialize(x, digits = 8, pretty = FALSE)

bricklayer_json_unserialize(txt)
```

Arguments

x	Object to serialise.
digits	Decimal digits retained for doubles (default 8, the jsonlite default). Raise it where full precision matters.
pretty	Indent the output.
txt	JSON produced by bricklayer_json_serialize().

Details

Use this, not bricklayer_json_to_json(), whenever the JSON has to reconstruct the object faithfully. bricklayer_json_to_json() writes the DATA – which is what an API or a human wants, and which loses factor levels, matrix dimensions, classes and every other attribute. These two keep them, at the cost of JSON no other tool will understand.

Value

bricklayer_json_serialize() returns a length-1 character vector of class json; bricklayer_json_unserialize() returns the original object.

See Also

bricklayer_json_to_json() for plain data JSON, core_sha256() for fingerprinting the result.

Examples

```
# A factor survives the round trip with its levels intact.
f <- factor(c("b", "a", "b"), levels = c("a", "b", "c"))
back <- bricklayer_json_unserialize(bricklayer_json_serialize(f))
identical(back, f)

# So does a matrix, with its dimensions.
m <- matrix(1:6, nrow = 2)
identical(bricklayer_json_unserialize(bricklayer_json_serialize(m)), m)

# Plain data JSON does not keep either, which is the trade-off.
bricklayer_json_to_json(f)

# Nested lists, names and NULLs round trip too.
x <- list(a = 1:3, b = list(c = "x", d = NULL), e = TRUE)
identical(bricklayer_json_unserialize(bricklayer_json_serialize(x)), x)

# The serialised form is JSON, so it can be pinned like any other text.
nchar(core_sha256(bricklayer_json_serialize(m)))
```

`rmbl_keyed_digest` *Keyed digest and constant-time comparison (C backend)*

Description

`core_hmac_sha256()` is HMAC-SHA-256 (RFC 2104): a digest computed under a secret key. The difference from a plain `core_sha256()` is AUTHENTICATION – anyone can recompute a SHA-256 and so anyone can forge one after editing a manifest, but only a holder of the key can produce a matching HMAC. This is what makes `capsule_sign()` 's `"hmac"` scheme meaningful.

Usage

```
core_hmac_sha256(key, message)

core_digest_equal(a, b)
```

Arguments

`key`	Secret key, as a length-1 character vector or a raw vector. Keys longer than the 64-byte block are hashed down first, per RFC 2104. Use at least 32 bytes of real entropy; against a quantum adversary Grover halves the effective key strength, so a 256-bit key retains a 128-bit margin.
`message`	Message to authenticate, as a length-1 character vector or a raw vector.
`a, b`	Digests to compare, as length-1 character vectors.

Details

`core_digest_equal()` compares two digests in constant time. Use it instead of `==` whenever the comparison is against a value an attacker supplied: a short-circuiting comparison leaks, through its own timing, how many leading characters were correct, which is enough to recover a tag byte by byte.

Value

`core_hmac_sha256()` a length-1 character vector: 64 lowercase hex characters. `core_digest_equal()` a length-1 logical; `FALSE` when the two differ in length.

References

Krawczyk H, Bellare M, Canetti R (1997). HMAC: Keyed-Hashing for Message Authentication. RFC 2104. doi:10.17487/RFC2104

Examples

```
# RFC 4231 test case 2.
core_hmac_sha256("Jefe", "what do ya want for nothing?")

# The key changes the digest, so a manifest cannot be re-signed
# without it.
core_hmac_sha256("key-a", "manifest")
core_hmac_sha256("key-b", "manifest")

# Raw keys and messages are accepted.
core_hmac_sha256(as.raw(rep(0x0b, 20)), "Hi There")

# Compare tags in constant time, never with ==.
tag <- core_hmac_sha256("k", "m")
core_digest_equal(tag, core_hmac_sha256("k", "m"))
core_digest_equal(tag, core_hmac_sha256("k", "tampered"))
core_digest_equal(tag, "too-short")
```

`rmbl_merkle` *Merkle tree over capsule chunks (C backend)*

Description

A single SHA-256 over a whole file tells you it changed. A Merkle tree over its chunks tells you WHICH chunk changed, and proves that one chunk belongs to the pinned file without re-reading the rest of it.

Usage

```
merkle_root(chunks)

merkle_leaves(chunks)

merkle_proof(chunks, index)

merkle_verify(leaf, proof, root)
```

Arguments

`chunks`	Character vector of chunk contents, in order. Use `chunk_file()` to produce it from a file.
`index`	1-based index of the chunk to prove.
`leaf`	The chunk whose membership is being verified.
`proof`	The list returned by `merkle_proof()`.
`root`	The expected root digest.

Details

merkle_root() reduces the chunks to one root digest. merkle_leaves() returns the per-chunk digests the root is built from, so two capsules can be diffed chunk by chunk. merkle_proof() returns the sibling digests on the path from one leaf to the root, and merkle_verify() replays that path.

An unpaired node at an odd level is PROMOTED unchanged rather than hashed against a duplicate of itself. Duplicating it would let two different chunk lists produce the same root – the weakness behind CVE-2012-2459 – so promotion is a correctness requirement, not a preference.

Value

merkle_root() a length-1 character vector (64 hex characters), or NA for no chunks. merkle_leaves() a character vector of per-chunk digests. merkle_proof() a list with sibling (character) and side ("left" / "right") . merkle_verify() a length-1 logical.

Examples

```
chunks <- c("row1,row2", "row3,row4", "row5,row6", "row7,row8")

root <- merkle_root(chunks)
root

# A single chunk's root is just its own digest.
merkle_root("only") == core_sha256("only")

# The leaves are the per-chunk digests, so a diff names the culprit.
before <- merkle_leaves(chunks)
after <- merkle_leaves(c(chunks[1:2], "row5,row6-EDITED", chunks[4]))
which(before != after)

# Prove chunk 3 belongs, without holding chunks 1, 2 or 4.
pr <- merkle_proof(chunks, 3)
pr
merkle_verify(chunks[3], pr, root)

# The proof fails for a chunk that was not in the tree.
merkle_verify("row5,row6-EDITED", pr, root)

# Any change to any chunk changes the root.
merkle_root(chunks) == merkle_root(c(chunks[1:3], "row7,row8 "))
```

---

rmbl_online *Exact summary statistics accumulated in blocks*

Description

An accumulator for the mean, variance, skewness and kurtosis of a column that does not fit in memory. Feed it blocks with summary_update() and read the result with summary_stats().

Usage

```
online_summary(x = NULL)

summary_update(acc, x)

summary_merge(a, b)

summary_stats(acc)
```

Arguments

x	Numeric vector: the first block, or NULL for an empty accumulator.
acc, a, b	Accumulators from online_summary().

Details

The merge is EXACT, not approximate: it uses Chan, Golub and LeVeque's parallel combination of central sums, extended to the third and fourth moments by Terriberry, so accumulating a column in blocks gives the same answer as core_moments() over the whole column at once. That is what makes it usable for a capsule member of any size: memory stays constant in the number of blocks.

The object is a plain list and is copied on assignment like any other R value, so update returns the new accumulator and you must keep it: acc <- summary_update(acc, block).

Value

online_summary(), summary_update() and summary_merge() return an object of class bricklayer_online; summary_stats() returns a named numeric with n, mean, variance, sd, skewness and kurtosis.

References

Chan TF, Golub GH, LeVeque RJ (1983). Algorithms for computing the sample variance: analysis and recommendations. *The American Statistician* 37(3), 242–247. doi:10.1080/00031305.1983.10483115

See Also

core_moments() for the single-pass batch version.

Examples

```
set.seed(1)
x <- stats::rnorm(1000)

# Accumulate in blocks of 100.
acc <- online_summary()
for (i in seq(1, 1000, by = 100)) {
  acc <- summary_update(acc, x[i:(i + 99)])
}
summary_stats(acc)

# Which is exactly the batch answer, not an approximation to it.
```

```
all.equal(summary_stats(acc)[["variance"]], stats::var(x))
all.equal(summary_stats(acc)[["mean"]], mean(x))

# Two accumulators built independently can be merged, so blocks can be
# processed in any order or on different machines.
a <- summary_update(online_summary(), x[1:400])
b <- summary_update(online_summary(), x[401:1000])
all.equal(summary_stats(summary_merge(a, b)), summary_stats(acc))

# An empty accumulator reports nothing rather than zero.
summary_stats(online_summary())
```

`rmbl_reservoir` *Uniform sample of a stream in one pass*

Description

Vitter's Algorithm R: draws `k` items from a stream, each item equally likely to be retained, in a single pass and using memory proportional to `k` rather than to the stream's length. Use it to take a fair sample of a capsule member you cannot hold, or whose length you do not know in advance.

Usage

```
reservoir_indices(n, k, seed = 42L)

reservoir_sample(x, k, seed = 42L)
```

Arguments

`n`	Length of the stream.
`k`	Sample size. Values above the stream length give the whole stream.
`seed`	Seed for the core's generator (default 42).
`x`	Vector to sample from.

Details

`reservoir_indices()` returns the retained positions, so the same sample can be applied to a file read line by line. `reservoir_sample()` applies it to a vector in memory.

The stream is sampled with the core's own generator seeded by `seed`, not R's, so the sample is reproducible and R's random stream is left alone – a capsule whose sample changed between runs would not be reproducible.

Value

`reservoir_indices()` a sorted numeric vector of 1-based positions; `reservoir_sample()` the corresponding elements of `x`.

References

Vitter JS (1985). Random sampling with a reservoir. *ACM Transactions on Mathematical Software* 11(1), 37–57. doi:10.1145/3147.3165

Examples

```
# A reproducible sample of 5 from 1000.
reservoir_indices(1000, 5, seed = 1)
reservoir_sample(letters, 4, seed = 2)

# Reproducible, and independent of R's own RNG.
identical(reservoir_indices(1000, 5, seed = 1),
          reservoir_indices(1000, 5, seed = 1))

# Asking for more than the stream holds returns the whole stream.
reservoir_indices(3, 10)

# Every item is equally likely: over many seeds the retained positions
# are spread uniformly rather than favouring the start or the end.
hits <- unlist(lapply(1:400, function(s) reservoir_indices(50, 5, s)))
round(mean(hits))          # near the midpoint, 25.5
```

---

rmbl_rule_library *Ready-made validation rules*

Description

Constructors for the checks most schemas need, each returning a `rule()` that `validate_rules()` evaluates. Use `rule()` directly for anything not covered here.

Usage

```
rule_in_set(column, set, na_pass = TRUE, severity = c("warning", "fatal"))

rule_between(column, lo, hi, na_pass = TRUE, severity = c("warning", "fatal"))

rule_not_null(column, severity = c("warning", "fatal"))

rule_unique(column, severity = c("warning", "fatal"))

rule_regex(column, pattern, na_pass = TRUE, severity = c("warning", "fatal"))

rule_increasing(column, strictly = FALSE, severity = c("warning", "fatal"))

rule_within_n_mads(
  column,
  n = 3,
  na_pass = TRUE,
```

```
  severity = c("warning", "fatal")
)

rule_complete_rows(severity = c("warning", "fatal"))

rule_distinct_rows(columns = NULL, severity = c("warning", "fatal"))

rule_col_count(n, severity = c("warning", "fatal"))
```

Arguments

column	Column the rule applies to.
set	Allowed values (rule_in_set) .
na_pass	Treat NA as passing (default TRUE; see above).
severity	"warning" (default) or "fatal".
lo, hi	Inclusive bounds (rule_between) .
pattern	Regular expression the values must match (rule_regex) .
strictly	Require a strict increase rather than non-decreasing (rule_increasing) .
n	Multiplier for rule_within_n_mads, or the expected count for rule_col_count.
columns	Columns that jointly must be unique (rule_distinct_rows) , or NULL for all of them.

Details

NA handling is explicit and per-rule, because the right answer differs. rule_not_null() exists precisely to fail on NA. The value rules (rule_in_set, rule_between, rule_regex, rule_within_n_mads) treat NA as PASSING by default, so that a column's missingness is reported once by rule_not_null() or max_missing_fraction rather than again by every other rule; set na_pass = FALSE to make them fail on it instead.

Value

A rule() object.

See Also

rule() for an arbitrary predicate, validate_rules() to evaluate them, infer_schema() for the structural checks that need no rules at all.

Examples

```
df <- data.frame(
  id = c(1, 2, 2),
  grade = c("a", "b", "z"),
  score = c(5, 200, 7),
  email = c("a@b.com", "nope", "c@d.org"),
  day = c(3, 1, 2),
  stringsAsFactors = FALSE
```

```
)

rules <- list(
  rule_unique("id", severity = "fatal"),
  rule_in_set("grade", c("a", "b", "c")),
  rule_between("score", 0, 100),
  rule_regex("email", "^[^@]+@[^@]+\\\\.[a-z]+$"),
  rule_increasing("day")
)
names(validate_rules(df, rules))

# Each names the rows that failed.
validate_rules(df, rules)$grade_in_set$rows

# Clean data passes every one of them.
ok <- data.frame(id = 1:3, grade = c("a", "b", "c"),
                 score = c(5, 50, 7),
                 email = c("a@b.com", "c@d.org", "e@f.net"),
                 day = 1:3, stringsAsFactors = FALSE)
length(validate_rules(ok, rules))

# A robust outlier rule: MADs from the median, not standard
# deviations from the mean, so one wild value cannot hide the others.
validate_rules(data.frame(v = c(1, 2, 3, 2, 1, 900)),
               rule_within_n_mads("v", 5))$v_within_mads$rows

# Whole-table rules.
validate_rules(df, rule_distinct_rows())
validate_rules(df, rule_col_count(5))
validate_rules(df, rule_complete_rows())
```

`rule` *Declare a validation rule*

Description

Builds a rule for `validate_rules()`: a predicate over one column, or over the whole data frame, with a severity and a message. Rules live alongside a schema in a provenance record, which keeps the project-specific checks in data rather than scattered through code.

Usage

```
rule(
  name,
  expr,
  column = NULL,
  severity = c("warning", "fatal"),
  message = NULL
)
```

Arguments

name	Short identifier for the rule.
expr	A function, as described above.
column	Column the rule applies to, or NULL for a table-level rule.
severity	"warning" (default) or "fatal".
message	Human-readable description of what a failure means. Defaults to a generated one naming the rule.

Details

expr is a function. Given column, it receives that column and must return a logical vector the same length (TRUE = the row passes) or a single logical for a whole-column property. Given no column, it receives the whole data frame and must return a single logical.

Value

A list of class bricklayer_rule.

See Also

validate_rules(), validate_schema() for the structural checks that need no rules.

Examples

```
# A column predicate, applied row-wise.
rule("age_non_negative", function(v) v >= 0, column = "age")

# A whole-column property.
rule("id_unique", function(v) !anyDuplicated(v), column = "id",
     severity = "fatal")

# A table-level rule spanning two columns.
rule("dates_ordered", function(df) all(df$start <= df$end))
```

---

sha256_file *Compute a File's SHA256 Digest*

Description

Returns the SHA256 digest of a file as a lowercase hex string, computed by the package's own compiled SHA-256 core (or, when the file is sourced standalone inside a capsule bundle, by the pure-R FIPS 180-4 implementation it ships). Used to record and verify data provenance.

Usage

```
sha256_file(path)
```

Arguments

path	Path to the file to hash.

Value

The SHA256 digest as a character string.

Examples

```
f <- tempfile()
writeLines("hello capsule", f)
sha256_file(f)

# Deterministic: the same bytes always yield the same digest.
identical(sha256_file(f), sha256_file(f))

# Any change to the file changes the digest (tamper-evidence).
before <- sha256_file(f)
writeLines("hello capsule (edited)", f)
after <- sha256_file(f)
before == after      # FALSE

# Provenance pin: record a digest, verify it later.
pinned <- sha256_file(f)
stopifnot(sha256_file(f) == pinned)
```

---

share *Share of a total, in percent*

Description

What fraction of a total each count is, with the Wilson interval.

Usage

```
share(x, ...)

## S3 method for class 'data.frame'
share(x, count, by = NULL, total = NULL, conf_level = 0.95, ...)

## Default S3 method:
share(x, total = NULL, conf_level = 0.95, ...)
```

Arguments

x	A data frame, or a numeric vector of counts.
...	Passed to methods.
count	Column of counts: non-negative whole numbers.

by	Character vector of grouping columns. Shares are computed over the groups, so they sum to 100 unless `total` says otherwise.
total	The denominator. `NULL` (default) sums `count`, which makes the shares sum to 100. A number, or a column name, uses that instead – for the case where some of the total is not in the table.
conf_level	Confidence level for the interval.

Details

A share and a rate are different quantities: a share's denominator is the total of the same events, so shares over a complete grouping sum to 100. Use `rate()` when the denominator is a population.

The interval is Wilson's, not the textbook normal approximation. The normal interval on a proportion is wrong in exactly the cases people reach for it – small counts and shares near 0 or 100, where it runs past the ends of the scale and reports a negative percentage. Wilson's stays inside the scale and is accurate at those counts.

Value

A data frame of class `rmbl_share` with the grouping columns, `count`, `total`, `share`, `lower` and `upper`, plus a `share` attribute recording the denominator and the confidence level.

See Also

`rate()` for events per population, `yoy()` for change between periods.

Examples

```
stops <- data.frame(
  division = c("North", "South", "East"),
  stops = c(412, 77, 3))

# shares of the table's own total, summing to 100
share(stops, stops, by = "division")

# East is 0.6% of stops, and the interval does not run below zero
# the way a normal approximation would
share(stops, stops, by = "division")$lower

# a denominator from outside the table
share(stops, stops, by = "division", total = 10000)
```

---

signing_public_key *Public half of a signing key*

Description

Strips the secret seed, leaving only what a verifier needs. Publish this; never the object returned by `pqc_keygen()`.

Usage

```
signing_public_key(key)
```

Arguments

key	A bricklayer_signing_key from pqc_keygen().

Value

A list of class bricklayer_public_key: root, pub_seed, height, scheme.

Examples

```
key <- pqc_keygen(height = 2)
pub <- signing_public_key(key)

# The secret seed is gone.
is.null(pub$sk_seed)

# And verification works from the public half alone.
sig <- capsule_sign("manifest-digest", key)
capsule_verify("manifest-digest", sig, pub)
```

---

sir *Standardised incidence ratio, with an exact interval*

Description

The ratio of observed to expected, with the exact Poisson interval for it. A ratio of one is the overall experience; above one is an excess.

Usage

```
sir(observed, expected, area = NULL, conf_level = 0.95)
```

Arguments

observed	Observed counts.
expected	Expected counts, from expected_counts().
area	Optional labels.
conf_level	Confidence level.

Details

The interval is the exact Poisson one, from the relation between the Poisson and gamma distributions, and so is identical to stats::poisson.test 's. It is the interval to use here because the counts that matter are small: a normal approximation on an observed count of three is not an interval, it is a decoration.

Value

A data frame with `observed`, `expected`, `sir`, `lower`, `upper` and `excess` – whether the interval excludes one.

References

Lawson, A. B. *Using R for Bayesian Spatial and Spatio-Temporal Health Modeling*. Chapman and Hall/CRC, Chapter 1.

See Also

`eb_rates()`, `funnel_limits()`

Examples

```
sir(observed = c(30, 12, 3), expected = c(20, 14, 5),
    area = c("North", "South", "East"))

# An observed count of three carries almost no information, and the
# interval says so rather than hiding it.
sir(3, 5)
```

---

stay_summary *Length of stay with its distribution and interval*

Description

Where `alos()` takes the total days and returns a mean, this takes one value per person and reports the spread as well, because a mean stay is a poor summary of a distribution that is usually skewed.

Usage

```
stay_summary(days_per_person, conf_level = 0.95)
```

Arguments

`days_per_person`	Days served, one element per person.
`conf_level`	Confidence level for the interval on the mean.

Details

The interval is the ordinary t interval on a mean. It describes uncertainty about the AVERAGE stay, not the spread of stays, and on a skewed distribution the median and the interquartile range say more about a typical stay than the mean does – which is why they are returned beside it rather than left to be asked for.

Lakner's caveat on `alos()` applies here too (1976, p.16-17): a person still held has an unfinished stay, so a window shorter than the longest stay biases the mean DOWNWARD.

Value

A one-row data frame: `n`, `total_days`, `mean`, `sd`, `median`, `iqr`, `max`, `se`, `lower`, `upper`.

See Also

`alos()`, `adp()`, `stock_flow()`

Examples

```
# a skewed distribution: most stays short, a few long
stays <- c(rep(1, 40), rep(3, 30), rep(10, 20), 60, 90, 120)
stay_summary(stays)

# the mean is pulled well above the median by the long tail
stay_summary(stays)[, c("mean", "median", "max")]
```

---

step_change *A single step change, with the scan's own null distribution*

Description

Scans every admissible split of the series, reports the one with the largest mean difference, and gives it a p-value from the permutation distribution of the MAXIMUM over splits – not from the best split's own test, which is the standard way to find a change point in noise.

Usage

```
step_change(y, x = NULL, min_segment = 2L, n_perm = 9999L, seed = 1L)
```

Arguments

`y`	The series, in period order.
`x`	The periods. Used only for labelling the break.
`min_segment`	Fewest periods either side of the break.
`n_perm`	Permutations for the null distribution. The exact enumeration is used instead when the series is short enough for it.
`seed`	Seed for the permutations, so the p-value is reproducible.

Value

A list with `break_after` (the period the series changes after), `index`, `before`, `after`, `difference`, `statistic`, `p_value`, `n_perm` and `method`.

References

The permutation distribution of the maximum over splits, rather than the chosen split's own test, is what makes this a test of whether there is a break rather than a way of locating the largest wobble. See any treatment of the change-point problem, e.g. Coles, S. *An Introduction to Statistical Modeling of Extreme Values* (Springer), which discusses change-point detection alongside the threshold choices that raise the same multiple-comparison issue.

Examples

```
# A clear step down after the third period.
step_change(c(100, 104, 98, 60, 63, 58))

# Pure noise: the best split is still found, and is not significant.
set.seed(2)
step_change(stats::rnorm(12))$p_value
```

---

stock_flow *Stock and flow side by side, with the decomposition*

Description

Reports the same person-days as a stock and as a flow, and says how much of any change in the stock came from the number of people and how much from how long they stayed.

Usage

```
stock_flow(
  days,
  people,
  period = NULL,
  t = 365,
  exposure = NULL,
  per = 1e+05,
  baseline = c("first", "previous")
)
```

Arguments

days	Person-days, one element per period.
people	Number of people, one element per period.
period	Optional labels for the periods.
t	Length of each period in days, one value or one per period. Default 365. period_days() turns dates into this.
exposure	Optional population to express rates against, one per period; for example provincial residents.
per	Rate denominator when exposure is given. Default 100000.

baseline — What the change columns compare against: "first" (the default) measures every period against the first, which is what a report on a whole window wants; "previous" measures each period against the one before it, which is what a series wants.

Details

The decomposition is exact, because days are people times length of stay: a change in days is $(1 + p)(1 + l) - 1$ for proportional changes p in people and l in stay. The two rates can therefore carry OPPOSITE signs, and the point of putting them in one table is that neither can then be quoted alone.

Value

A data frame of class rmbl_stock_flow, one row per period: people, days, alos, adp, and when exposure is supplied flow_rate and stock_rate. Change columns compare each period with the first.

References

Lakner, E. (1976) *A Manual of Statistical Sampling Methods for Corrections Planners*. University of Illinois at Urbana-Champaign.

See Also

adp(), alos()

Examples

```
# Fewer people, held longer: the flow falls while the stock rises.
stock_flow(days = c(115674, 126121), people = c(12647, 9608),
           period = c("2023", "2025"),
           exposure = c(15495050, 16256538))
```

---

timestamp_verify *Verify an RFC 3161 timestamp token*

Description

Checks that a timestamp token covers the bytes given, reports the time it asserts, and verifies the timestamping authority's signature under a certificate you supply.

Usage

```
timestamp_verify(
  token,
  data,
  certificate = NULL,
  trust = NULL,
  crls = list(),
  at_time = NULL
)

timestamp_info(token)
```

Arguments

token	The token: a raw vector, or a path to a `.tsr` / `.tst` file. A full `TimeStampResp` or a bare `TimeStampToken` are both accepted.
data	The bytes the token should cover: a raw vector, or a path to a file.
certificate	The authority's certificate, DER or PEM, as raw or a path. Optional: when the token embeds a certificate, that one is used and the fact is reported.
trust	Trust anchors for validating that certificate – see `cert_chain_verify()`. Without them the signature is still checked, but nothing vouches for the key that made it.
crls	Optional CRLs to check the chain against, as raw vectors or paths.
at_time	The time to check certificate validity at. Defaults to the time the token asserts, which is usually what is wanted.

Details

Pass `trust` and the certificate is validated too: the chain is built to an anchor you name, every signature in it is verified, every validity window is checked, an issuer must be a CA, and the leaf must carry the timeStamping extended key usage. Omit `trust` and the signature is still checked but `certificate_trust` is reported as failed, because without an anchor a passing signature says only that the key in the certificate signed the token – not that anyone should believe that certificate.

Validity windows are checked at the time the TOKEN asserts, unless `at_time` says otherwise. A token signed in 2020 under a certificate that expired in 2021 was validly signed, and judging it by today's date would reject it for a reason unconnected to its validity.

RSA and ECDSA over the NIST prime curves P-256, P-384 and P-521 are verified. Anything else – a post-quantum signature, a compressed EC point, an Edwards curve – is reported as unverifiable rather than treated as valid, which is the safe direction for a verifier.

Still not done: name constraints, policy mapping, and fetching revocation data. A CRL has to be handed in through `crls`; nothing is retrieved over the network.

Value

A list of class `bricklayer_timestamp`: `ok`, `time` (a `POSIXct` in UTC), `serial`, `policy`, `hash_algorithm`, `signature_algorithm`, and a `checks` data frame.

References

Adams, C., Cain, P., Pinkas, D., and Zuccherato, R. (2001). Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP). RFC 3161.

See Also

`chain_seal()` for the ordering a timestamp cannot give, `capsule_attest()` for authorship.

Examples

```
# Tokens come from a timestamping authority, so there is nothing to
# demonstrate offline; this is the shape of the call.
## Not run:
res <- timestamp_verify("response.tsr", data = "manifest.json")
res$ok
res$time

## End(Not run)
```

`top_correlations` *Strongest pairwise correlations in a data frame*

Description

Ranks the numeric column pairs by the strength of their association, so a wide table's structure can be read without squinting at a correlation matrix.

Usage

```
top_correlations(data, n = 10L, method = c("spearman", "pearson"), min_abs = 0)
```

Arguments

`data`	A data frame; non-numeric columns are ignored.
`n`	Number of pairs to return (default 10).
`method`	`"spearman"` (default) or `"pearson"`.
`min_abs`	Report only pairs whose absolute correlation reaches this (default 0).

Details

Spearman is the default deliberately. Pearson measures LINEAR association only, so it understates a relationship that is perfectly monotone but curved, and a single outlier can manufacture or destroy it. On data you have not yet inspected – which is the situation this function is for – the rank correlation is the safer question to ask.

Value

A data frame of class `bricklayer_correlations` with `x`, `y`, `correlation` and `abs_correlation`, strongest first.

See Also

`core_cor_spearman()`, `core_cov()`

Examples

```
set.seed(1)
n <- 200
df <- data.frame(
  a = stats::rnorm(n),
  b = stats::rnorm(n),
  grade = sample(letters[1:3], n, TRUE)
)
df$c <- df$a * 2 + stats::rnorm(n, sd = 0.1)   # strongly related to a
df$d <- exp(df$a)                              # monotone but curved

top_correlations(df)

# The curved pair is ranked correctly by Spearman and understated by
# Pearson, which is why Spearman is the default.
tc <- top_correlations(df, method = "spearman")
tc[tc$x == "a" & tc$y == "d", "correlation"]
tp <- top_correlations(df, method = "pearson")
tp[tp$x == "a" & tp$y == "d", "correlation"]

# Filter to the pairs worth looking at.
top_correlations(df, min_abs = 0.5)
```

---

to_ascii *Transliterate Text to Plain ASCII*

Description

Converts a character vector to plain 7-bit ASCII, transliterating accented or non-Latin characters to their nearest ASCII equivalent (for example, an accented capital A becomes a plain "A"). Falls back to dropping any character that has no transliteration. This is the deterministic "fallback" used by `ascii_fallback()`.

Usage

```
to_ascii(x)
```

Arguments

x	A character vector.

Value

A character vector containing only ASCII characters.

Examples

```
# Latin accents fold to their nearest ASCII letter.
to_ascii("Prof. \u00c1ngela Zorro Medina")  # "Prof. Angela Zorro Medina"

# Vectorised over the input.
to_ascii(c("Se\u00e1n", "Zo\u00eb", "na\u00efve"))

# Non-Latin scripts are romanised when stringi is available.
if (requireNamespace("stringi", quietly = TRUE))
  to_ascii("\u041c\u043e\u0441\u043a\u0432\u0430")  # "Moskva" (Cyrillic)

# Either way the result is guaranteed pure 7-bit ASCII (never "?").
all(charToRaw(to_ascii("caf\u00e9")) < 128)
```

---

trend_test *Trend in a short series*

Description

The Mann-Kendall rank test for monotone trend with the Theil-Sen median-of-slopes estimator: no distributional assumption, resistant to a single aberrant period, and meaningful at the series lengths an annual administrative extract actually has.

Usage

```
trend_test(
  y,
  x = NULL,
  value = NULL,
  period = NULL,
  exact = NULL,
  alternative = c("two.sided", "increasing", "decreasing"),
  conf_level = 0.95
)
```

Arguments

y	The series, in period order, or a data frame.
x	The periods. Defaults to the position, which is right for an evenly spaced series.
value, period	Column names, when y is a data frame.
exact	Whether to compute the exact null distribution of Mann-Kendall's S by enumeration. Feasible and used by default up to n = 8 (40,320 orderings); above that the normal approximation with the tie and continuity corrections is used.

alternative	"two.sided", "increasing" or "decreasing".
conf_level	Confidence level for the slope interval.

Details

Kendall's tau here is S over the number of comparable pairs, so it is the rank correlation between the value and the period.

The slope interval is the standard rank-based one: the pairwise slopes are sorted and the interval runs between the order statistics that Mann-Kendall's variance places at the chosen level, so it is consistent with the test rather than derived from a different model.

Value

A list with S, tau, p_value, slope (Theil-Sen), intercept, slope_lower / slope_upper (the distribution-free interval), n and method.

References

Sen, P. K. (1968). Estimates of the regression coefficient based on Kendall's tau. *Journal of the American Statistical Association* 63(324), 1379-1389, for the median-of-slopes estimator and the distribution-free interval.

Wilcox, R. R. *Modern Statistics for the Social and Behavioral Sciences: A Practical Introduction* treats Theil-Sen among the regression methods that carry no normality assumption, which is the reason for preferring it on a series this short.

See Also

step_change(), count_trend()

Examples

```
# Five years of placements.
y <- c(402, 377, 190, 268, 331)
trend_test(y)

# A monotone series is detected even at n = 5, where a regression's
# standard error would be nearly uninformative.
trend_test(c(1, 2, 3, 4, 5))

# One aberrant period does not create a trend.
trend_test(c(100, 100, 100, 100, 900))$p_value

# From a data frame.
d <- data.frame(year = 2019:2023, n = y)
trend_test(d, value = "n", period = "year")$slope
```

validate_rules *Apply declared rules to a data frame*

Description

Evaluates each rule from rule() and returns the failures in the same shape validate_schema() uses, so the two can be combined and handed to apply_schema_validation() together.

Usage

```
validate_rules(data, rules)
```

Arguments

data	A data frame.
rules	A list of rule() objects, or a single rule.

Details

A rule whose column is absent is SKIPPED rather than failed – a missing column is validate_schema() 's business, and reporting it twice buries the real finding. A rule that ERRORS is reported as a failure naming the error, never swallowed: a rule that cannot run has not passed.

Value

A named list of issues, each with severity, message, and for a row-wise rule n_failed and rows (the first failing row indices). Empty when everything passes.

See Also

rule(), validate_schema(), apply_schema_validation()

Examples

```
df <- data.frame(id = c(1, 2, 2), age = c(30, -5, 40),
                 start = c(1, 5, 3), end = c(2, 4, 9))

rules <- list(
  rule("age_non_negative", function(v) v >= 0, column = "age"),
  rule("id_unique", function(v) !anyDuplicated(v), column = "id",
       severity = "fatal"),
  rule("dates_ordered", function(d) all(d$start <= d$end))
)

issues <- validate_rules(df, rules)
names(issues)

# A row-wise failure reports how many rows and which.
issues$age_non_negative$n_failed
```

```
issues$age_non_negative$rows

# Clean data produces nothing.
clean <- data.frame(id = 1:3, age = c(30, 31, 40), start = 1:3, end = 4:6)
length(validate_rules(clean, rules))

# A rule for an absent column is skipped, not failed -- a missing
# column is validate_schema()'s finding to report, not this one's.
length(validate_rules(data.frame(id = 1:3), rules[1:2]))

# A rule that errors is a failure, not a silent pass.
broken <- list(rule("bad", function(v) stop("boom"), column = "age"))
validate_rules(clean, broken)$bad$message
```

`validate_schema` *Validate a Data Frame Against a Provenance Schema*

Description

Checks a raw data frame against the `schema` block of a provenance object and returns the issues found rather than raising, so the caller decides how to react. `apply_schema_validation()` is the wrapper that turns them into errors and warnings.

Usage

```
validate_schema(df_raw, provenance)
```

Arguments

`df_raw`	The data frame to validate.
`provenance`	A provenance list as returned by `load_provenance()`, or `list(schema = infer_schema(df))`. The `schema` block may contain: • `expected_columns` – required column names. A missing one is `"fatal"`; everything else below is a `"warning"`. • `structural_invariants` – `min_data_rows` and `max_data_rows`. • `expected_value_sets` – a named list of allowed values per column. • `expected_types` – a named character vector of expected classes. `integer`, `numeric` and `double` are treated as interchangeable, since a re-release legitimately widens one to another. • `numeric_ranges` – a named list of `c(min =, max =)` bounds. • `max_missing_fraction` – a named numeric of per-column ceilings on the share of `NA`. `infer_schema()` produces all six from data you already trust.

Details

Every schema field is optional and is checked only when present, so a schema written for an earlier version keeps working unchanged.

This is the STRUCTURAL check – names, types, bounds, value sets. It cannot tell you that a column kept its name, type and range while its distribution moved; `capsule_drift()` answers that.

Value

A named list of issues; each issue is a list with `severity` (`"fatal"` or `"warning"`) and a human-readable `message`. A zero-length list means the data frame is clean.

See Also

`infer_schema()` to derive a schema, `capsule_drift()` for the distributional check, `apply_schema_validation()` to raise on the issues.

Examples

```
prov <- list(schema = list(
  expected_columns      = c("id", "year"),
  structural_invariants = list(min_data_rows = 1),
  expected_value_sets   = list(year = 2020:2025)
))
df <- data.frame(id = 1:3, year = c(2020, 2021, 2030))
issues <- validate_schema(df, prov)
names(issues)  # flags the out-of-set year value

# A column that silently changed type is caught.
typed <- list(schema = list(expected_types = c(id = "integer")))
validate_schema(data.frame(id = c("1", "2")), typed)[[1]]$message

# So is a value outside its pinned range, and excess missingness.
ranged <- list(schema = list(numeric_ranges = list(v = c(min = 0, max = 1))))
validate_schema(data.frame(v = c(0.5, 9)), ranged)[[1]]$message

gappy <- list(schema = list(max_missing_fraction = c(v = 0.1)))
validate_schema(data.frame(v = c(1, NA, NA, 4)), gappy)[[1]]$message

# A clean frame produces nothing.
length(validate_schema(data.frame(id = 1:3, year = 2021), prov))
```

`verify_capsule` *Re-Verify an Entire Reproducible Data Capsule*

Description

Runs the full custody chain over a capsule directory in one call: the provenance manifest is readable, the pinned data file exists and matches its recorded `sha256` (and `size_bytes` / row count where recorded), the schema still validates, a recorded analysis script still matches its pinned hash, and every numeric cross-check stored in a results manifest still reproduces its recorded `PASS` / `DIFFER` status from its own `observed` / `expected` / `tol` fields.

Usage

```
verify_capsule(
  capsule_dir,
  provenance_file = "data_provenance.json",
  data_file = NULL,
  manifest_file = NULL,
  script_file = NULL
)
```

Arguments

`capsule_dir`	Directory containing the capsule.
`provenance_file`	Provenance JSON filename inside `capsule_dir` (default `"data_provenance.json"`) .
`data_file`	Data filename inside `capsule_dir`. Defaults to the provenance's `resource$filename`.
`manifest_file`	Optional results-manifest JSON (as written by `write_manifest_json()`) inside `capsule_dir`; checked when present.
`script_file`	Optional analysis-script filename inside `capsule_dir`; compared against the manifest's recorded `meta$script_sha256` when both are present.

Details

Entirely offline: nothing is downloaded and nothing is written.

Value

A list with `ok` (logical scalar: every check passed) and `checks` (data.frame with columns `check`, `ok`, `detail`) .

Examples

```
dir <- file.path(tempdir(), "capsule-example")
dir.create(dir, showWarnings = FALSE)
write.csv(data.frame(id = 1:3), file.path(dir, "d.csv"), row.names = FALSE)
prov <- list(resource = list(filename = "d.csv",
                             sha256 = sha256_file(file.path(dir, "d.csv"))))
writeLines(bricklayer_json_to_json(prov, auto_unbox = TRUE),
           file.path(dir, "data_provenance.json"))
verify_capsule(dir)$ok
```

verify_sha256 *Verify a File's SHA256 Against an Expected Digest*

Description

Computes the SHA256 digest of a file with the package's own compiled SHA-256 core and compares it to the expected value pinned in provenance.

Usage

```
verify_sha256(path, expected_sha)
```

Arguments

path	Path to the file to hash.
expected_sha	The expected SHA256 digest, as a lowercase hex string.

Value

A list with actual (computed digest), expected (the value passed in), and match (logical; TRUE if they are identical).

Examples

```
f <- tempfile()
writeLines("hello capsule", f)

# Matching digest -> match TRUE.
chk <- verify_sha256(f, sha256_file(f))
chk$match          # TRUE

# A wrong expected digest -> match FALSE, with both values reported.
bad <- verify_sha256(f, strrep("0", 64L))
bad$match          # FALSE
bad$actual         # the real digest
```

wayback_snapshot_url *Resolve a Wayback Machine snapshot URL*

Description

Queries the Internet Archive availability API (http://archive.org/wayback/available) for the closest archived snapshot of url and returns a directly-downloadable snapshot URL, or NULL if no snapshot exists or the lookup fails. This is the shared fetch failsafe the wider morie package family relies on: callers attempt the live source first and fall back to this snapshot when the source is unreachable.

Usage

```
wayback_snapshot_url(url, timestamp = NULL)
```

Arguments

`url`	The original source URL to look up.
`timestamp`	Optional 14-digit `YYYYMMDDhhmmss` target; the API returns the snapshot closest to it. Defaults to the most recent.

Value

A character scalar snapshot URL, or `NULL`.

Examples

```
wayback_snapshot_url("https://www.r-project.org/")
```

`wayback_snapshot_url_native` *Resolve a Wayback Machine snapshot URL (C++/libcurl)*

Description

Queries the Internet Archive "available" API for the closest archived snapshot of `url`. C++ backend; supersedes the older R-level resolver `wayback_snapshot_url`, which is kept for the pure-R path and now parses with the package's own JSON codec.

Usage

```
wayback_snapshot_url_native(url, timeout = 30L)
```

Arguments

`url`	URL to resolve.
`timeout`	Request timeout, seconds.

Value

The https snapshot URL, or `NULL` if none is archived.

Examples

```
# Input is validated before any network call:
try(wayback_snapshot_url_native(""))     # empty url -> error


# Uses the live Wayback service; degrades gracefully offline: any
# network failure returns NULL rather than erroring.
# Closest archived snapshot of a live page (or NULL if none archived).
wayback_snapshot_url_native("https://www.r-project.org/")

# A shorter timeout for a quick lookup.
wayback_snapshot_url_native("https://cloud.r-project.org/", timeout = 10)

# A never-archived URL returns NULL rather than erroring.
wayback_snapshot_url_native("https://example.invalid/never-archived")
```

---

write_manifest_json *Write a Manifest to JSON*

Description

Serializes a manifest to a pretty-printed JSON file with the native JSON codec (bricklayer_json_to_json()) ; no jsonlite needed.

Usage

```
write_manifest_json(manifest, path, canonical = FALSE)
```

Arguments

manifest	A manifest as returned by make_manifest() / built up with record().
path	Destination path for the JSON file.
canonical	Write the canonical form rather than the pretty-printed one: one line, keys sorted, which is what manifest_digest() hashes. Use it when the file itself has to be byte-stable rather than read by a person.

Value

The path, returned invisibly.

Examples

```
man <- make_manifest(list(project = "demo"), environment = FALSE)
man <- record(man, "row_count", observed = 20, expected = 20)
path <- write_manifest_json(man, tempfile(fileext = ".json"))
file.exists(path)
```

```
# Round-trips back through the package's own codec.
back <- bricklayer_json_from_json(path, simplifyVector = FALSE)
back$results$row_count$status        # "PASS"
```

`write_summary_txt` *Write a Plain-Language Run Summary*

Description

Writes a human-readable `SUMMARY.txt` into the output directory, covering run metadata, the exact absolute paths used, result counts, the files produced, and optional notes, contact, and licence lines.

Usage

```
write_summary_txt(
  manifest,
  output_dir,
  paths,
  what_was_done = NULL,
  contact = NULL,
  licence = NULL
)
```

Arguments

`manifest`	A manifest as returned by `make_manifest()`; its `meta` supplies project/author/run details.
`output_dir`	Directory to write `SUMMARY.txt` into and to list produced files from.
`paths`	A named list of absolute paths to report (e.g. `capsule`, `input`, `results`, `analysis_script`, `provenance`) .
`what_was_done`	Optional character vector of bullet points describing what the run did.
`contact`	Optional contact string appended to the summary.
`licence`	Optional licence string appended to the summary.

Value

The path to the written `SUMMARY.txt`, returned invisibly.

Examples

```
man <- make_manifest(list(project = "demo", author = "A. Author"),
                     environment = FALSE)
man <- record(man, "row_count", observed = 20, expected = 20)
out <- file.path(tempdir(), "demo-run")
dir.create(out, showWarnings = FALSE)
s <- write_summary_txt(man, out, paths = list(results = out))
readLines(s)[7:8]
```

write_text_fallback *Write Text as UTF-8, Falling Back to ASCII on an Encoding Error*

Description

Writes `text` to `path` as UTF-8. If the write raises an encoding error (for example a destination or locale that cannot represent the characters), it retries with an ASCII transliteration produced by `to_ascii()` so capsule generation never fails on a non-ASCII name.

Usage

```
write_text_fallback(text, path)
```

Arguments

`text`	A character vector of lines to write.
`path`	Destination file path.

Value

Invisibly, `path`.

Examples

```
p <- write_text_fallback(c("line one", "line two"),
                         tempfile(fileext = ".txt"))
readLines(p)
```

yoy *Year-over-year (and period-over-period) change*

Description

Computes the change from one period to the period `lag` places before it, matched on the period's own value rather than on row order, and carries an exact interval for count data.

Usage

```
yoy(x, ...)

## S3 method for class 'data.frame'
yoy(
  x,
  value,
  period,
  by = NULL,
```

```
  lag = 1L,
  fun = sum,
  units = c("count", "continuous", "percent"),
  min_base = NULL,
  conf_level = 0.95,
  direction = c("neutral", "higher_is_better", "lower_is_better"),
  complete = TRUE,
  ...
)

## S3 method for class 'numeric'
yoy(x, period = seq_along(x), ...)

## S3 method for class 'integer'
yoy(x, period = seq_along(x), ...)

## S3 method for class 'ts'
yoy(x, lag = NULL, ...)

## S3 method for class 'rmbl_yoy'
print(x, digits = 1L, palette = "diverging", color = NULL, n = 30, ...)
```

Arguments

x	An rmbl_yoy object.
...	Ignored.
value	For a data frame, the column holding the measure, as a string or a bare name.
period	For a data frame, the column holding the period (a year, a Date, a fiscal-year integer, an ordered factor). For a numeric vector, the periods themselves.
by	Optional grouping columns, as a character vector. The change is computed within each group.
lag	How many periods back to compare with. 1 is year-over-year on annual data; for a ts the default follows the series' own frequency, so monthly data compares with the same month a year earlier.
fun	Aggregation applied to value within a period and group, when there is more than one row. Default sum(), which is what a count needs.
units	What the measure is. "count" gets the exact rate-ratio interval. "continuous" gets the percent change without one, since a single pair of totals carries no information about its own variability. "percent" reports a percentage-POINT change and withholds the percent change, which for a percentage is a different quantity.
min_base	Smallest previous-period value for which a percent change is reported. Below it the percent is NA and the reason is recorded, rather than a large number that describes the denominator. Defaults to 20 for counts and to no gate otherwise.
conf_level	Confidence level for the count interval.

direction	Which way is an improvement: "higher_is_better", "lower_is_better" (segregation days, deaths in custody, use of force), or "neutral". Affects colour and the verdict column only, never the arithmetic.
complete	Whether to insert the missing periods in the observed range so that a gap is visible as a gap instead of closing up.
digits	Digits for the percent column.
palette	One of yoy_palettes().
color	Whether to emit ANSI colour. Defaults to colour only when writing to a terminal that has it, so a redirected or captured output stays plain text.
n	Maximum rows to print.

Value

An rmbl_yoy object: a data frame with one row per period (and group), and columns period, value, previous, change, pct_change (or pp_change for percentages), pct_lower and pct_upper for counts, verdict, and flag recording why a percent was withheld.

References

The exact interval for a ratio of two counts is the conditional-binomial (Clopper-Pearson) one, which is the construction stats::poisson.test uses and which this is verified against. On the reporting conventions, the Toronto Police Service's *Understanding Strip Searches in 2020 Methodological Report* – in the local corpus – reports year-over-year change on exactly this kind of administrative extract, and is the shape this function is built for.

See Also

yoy_html(), yoy_pdf(), yoy_summary()

Examples

```
seg <- data.frame(
  EndFiscalYear = rep(2019:2023, each = 2),
  Gender = rep(c("Female", "Male"), 5),
  Number_Of_Placements = c(31, 402, 28, 377, 12, 190, 19, 268, 24, 331)
)

y <- yoy(seg, value = "Number_Of_Placements", period = "EndFiscalYear",
         by = "Gender", direction = "lower_is_better")
y

# The interval is exact, so a small group does not get a confident
# percent it has not earned.
subset(as.data.frame(y), Gender == "Female")

# A percentage is handled as percentage points, not as a percent of a
# percent.
rate <- data.frame(year = 2019:2023, share = c(4.1, 4.6, 5.2, 5.0, 5.4))
yoy(rate, value = "share", period = "year", units = "percent")
```

yoy_delim *Write a change table as delimited text, JSON or Markdown*

Description

`yoy_csv()` separates fields with a comma and quotes any field that contains one; `yoy_tsv()` separates with a tab and replaces any tab inside a field, since a tab-separated field cannot contain one and writing it would shift every column after it.

Usage

```
yoy_csv(x, file, digits = NULL, na = "", metadata = TRUE, comment = "#", ...)

yoy_tsv(x, file, digits = NULL, na = "", metadata = TRUE, comment = "#", ...)

yoy_json(x, file, digits = NULL, pretty = TRUE, ...)

yoy_markdown(x, file, digits = 1L, align = TRUE, ...)
```

Arguments

`x`	An `rmbl_yoy` object from `yoy()`.
`file`	Output path, or `NULL` to return the text instead of writing it.
`digits`	Digits for the percent columns. `NULL`, the default, writes them unrounded, which is what a downstream calculation wants; give a number for a figure meant to be read.
`na`	How to write a missing value. The default empty string is what most readers expect; `"NA"` keeps R's own spelling.
`metadata`	Whether to lead with comment lines recording the lag, units, confidence level, base gate and direction. On by default, because a percent column means different things under different settings and the file is the only place a later reader can look. Markdown and JSON carry the same information structurally.
`comment`	Comment prefix for the metadata lines.
`...`	Ignored.
`pretty`	Whether to indent the JSON.
`align`	Whether to pad the Markdown columns so the source table is readable unrendered.

Value

The path, invisibly; or the text, when `file` is `NULL`.

Examples

```
d <- data.frame(year = 2019:2023, n = c(402, 377, 190, 268, 331))
y <- yoy(d, value = "n", period = "year")

# Straight to text, for inspection.
cat(yoy_csv(y, NULL))

# Markdown, for a report.
cat(yoy_markdown(y, NULL))

# JSON keeps the settings as fields rather than as comments.
substr(yoy_json(y, NULL), 1, 80)
```

`yoy_label` *Label a period without changing it*

Description

Attaches display labels to a change table. The arithmetic already ran on the period's value, so a label can only affect what is printed: relabelling cannot move a number.

Usage

```
yoy_label(x, labels)
```

Arguments

`x`	An `rmbl_yoy` object from `yoy()`.
`labels`	Either a function applied to the period column, or a character vector the same length as the number of distinct periods, or a named character vector mapping a period (as a string) to its label.

Value

The object, with a `period_label` column and the labels used by `print()`, `yoy_html()`, `yoy_pdf()` and the delimited writers.

See Also

`fiscal_year_label()`

Examples

```
seg <- data.frame(EndFiscalYear = 2019:2023,
                  n = c(402, 377, 190, 268, 331))
y <- yoy(seg, value = n, period = EndFiscalYear)

# 2023 means the fiscal year 2022/23, so say so.
```

```
yoy_label(y, fiscal_year_label)

# Any function will do.
yoy_label(y, function(p) paste0("FY", substr(p, 3, 4)))

# Or an explicit mapping, for the periods that need one.
yoy_label(y, c("2020" = "2019/20 (COVID)"))
```

`yoy_palettes` *Colour palettes for change tables*

Description

Colour palettes for change tables

Usage

```
yoy_palettes()
```

Value

A character vector of palette names accepted by `yoy_html()`, `yoy_pdf()` and `print()`.

Examples

```
yoy_palettes()
```

`yoy_render` *Render a change table to HTML or PDF*

Description

Render a change table to HTML or PDF

Usage

```
yoy_html(
  x,
  file,
  title = "Year-over-year change",
  subtitle = NULL,
  palette = "diverging",
  digits = 1L,
  bars = TRUE,
  interval = TRUE,
  notes = NULL,
```

```
  ...
)

yoy_pdf(
  x,
  file,
  title = "Year-over-year change",
  subtitle = NULL,
  palette = "diverging",
  digits = 1L,
  interval = TRUE,
  notes = NULL,
  width = 11,
  height = 8.5,
  ...
)
```

Arguments

x	An rmbl_yoy object from yoy().
file	Output path. yoy_html() writes a single self-contained HTML file with no external requests; yoy_pdf() writes a PDF using R's own device, so neither needs a package beyond base R.
title	Heading for the page.
subtitle	Optional line under the heading. Defaults to the table's own settings – lag, units, interval and base gate – so the reader can see what the percentages mean.
palette	One of yoy_palettes().
digits	Digits for the percent column.
bars	Whether to draw an in-cell bar proportional to the change, scaled to the largest absolute change in the table.
interval	Whether to show the exact interval column for counts.
notes	Optional character vector of footnotes.
...	Ignored.
width, height	PDF page size in inches.

Value

The path, invisibly.

Examples

```
d <- data.frame(year = rep(2019:2023, each = 2),
                region = rep(c("North", "South"), 5),
                n = c(31, 402, 28, 377, 12, 190, 19, 268, 24, 331))
y <- yoy(d, value = "n", period = "year", by = "region",
         direction = "lower_is_better")
```

```
h <- file.path(tempdir(), "change.html")
yoy_html(y, h, title = "Placements by region")
file.exists(h)

p <- file.path(tempdir(), "change.pdf")
yoy_pdf(y, p, title = "Placements by region")
file.exists(p)

unlink(c(h, p))
```

`yoy_summary` *Summarise a change table*

Description

Summarise a change table

Usage

```
yoy_summary(object, ...)

## S3 method for class 'rmbl_yoy'
yoy_summary(object, ...)
```

Arguments

`object`	An `rmbl_yoy` object.
`...`	Ignored.

Value

A data frame with one row per group giving the first and last period, the total change across the span, the compound annual growth rate, and how many periods moved each way.

Examples

```
d <- data.frame(year = 2018:2023, n = c(120, 131, 98, 140, 155, 149))
yoy_summary(yoy(d, value = "n", period = "year"))
```

yoy_write *Write a change table to a file, in whatever format the name implies*

Description

Write a change table to a file, in whatever format the name implies

Usage

```
yoy_write(x, file, format = "auto", ...)
```

Arguments

x	An rmbl_yoy object from yoy().
file	Output path.
format	Output format. "auto" reads it from the file extension: .html / .htm, .pdf, .csv, .tsv / .tab, .json, .md / .markdown.
...	Passed to the format's own writer, so the colour, digit and layout options of yoy_html() and yoy_pdf() are available here too.

Value

The path, invisibly.

See Also

yoy_html(), yoy_pdf(), yoy_csv()

Examples

```
d <- data.frame(year = 2019:2023, n = c(402, 377, 190, 268, 331))
y <- yoy(d, value = "n", period = "year")

for (ext in c("csv", "tsv", "json", "md", "html")) {
  f <- file.path(tempdir(), paste0("change.", ext))
  yoy_write(y, f)
  cat(ext, file.size(f), "bytes\n")
  unlink(f)
}
```

Index