Package: artoo (via r-universe)

October 24, 2026

Title Lossless CDISC-Native Input and Output for Clinical Datasets

Version 0.2.0

Description Reads and writes clinical-trial datasets losslessly across 'SAS' XPORT (XPT), Clinical Data Interchange Standards Consortium (CDISC) Dataset-JSON, and 'Apache Parquet', applying a specification to produce submission-ready Study Data Tabulation Model (SDTM) and Analysis Data Model (ADaM) datasets. A single canonical metadata model carries labels, CDISC data types, lengths, 'SAS' display formats, controlled-terminology references, and sort keys identically across every format, so conversion between any two formats is lossless by construction. Pure 'R' and lightweight, with no external 'SAS' or 'Java' runtime. Reads and writes CDISC Define-XML, the specification document that accompanies a submission, and renders it as HTML through the bundled Define-XML stylesheet. Implements the published format specifications for CDISC Dataset-JSON (<https://cdisc-org.github.io/DataExchange-DatasetJson/doc/dataset-json1-1.html>), CDISC Define-XML (<https://www.cdisc.org/standards/data-exchange/define-xml>), and 'SAS' XPORT (<https://www.loc.gov/preservation/digital/formats/fdd/fdd000466.shtml>).

License MIT + file LICENSE

URL https://vthanik.github.io/artoo/, https://github.com/vthanik/artoo

BugReports https://github.com/vthanik/artoo/issues

Depends R (>= 4.2)

Imports cli, hms, jsonlite, nanoparquet (>= 0.3.0), rlang (>= 1.1.0), S7, utf8

Suggests callr, digest, quarto, readxl, testthat (>= 3.0.0), tibble, withr, writexl, xml2, xslt

VignetteBuilder quarto

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Copyright file COPYRIGHTS

Encoding UTF-8

Language en-US

LazyData true

Roxygen list(markdown = TRUE)

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

Date/Publication 2026-08-25 14:45:14 UTC

RemoteUrl https://github.com/vthanik/artoo

RemoteRef HEAD

RemoteSha 6d1a35f643f908507264de046adb4d740c5216f8

Contents

apply_spec . . . 3
artoo_checks . . . 6
artoo_encodings . . . 9
artoo_formats . . . 10
artoo_spec . . . 11
cdisc_adae . . . 14
cdisc_adsl . . . 14
cdisc_dm . . . 15
cdisc_spec . . . 15
cdisc_specs . . . 16
cdisc_suppdm . . . 17
cdisc_ts . . . 18
cdisc_vs . . . 18
check_spec . . . 19
check_study . . . 20
columns . . . 22
conformance . . . 23
decode_column . . . 25
get_meta . . . 27
is_artoo_checks . . . 28
is_artoo_meta . . . 29
is_artoo_spec . . . 30
lint_define . . . 31
members . . . 32
read_dataset . . . 34
read_json . . . 35
read_ndjson . . . 36
read_parquet . . . 38
read_rds . . . 39
read_spec . . . 40

read_xpt 42
repair_spec 44
set_meta 45
set_type 46
spec_codelists 48
spec_comments 49
spec_datasets 50
spec_documents 51
spec_keys 52
spec_methods 53
spec_standard 54
spec_study 55
spec_variables 56
sync_meta 57
validate_define 58
validate_spec 60
write_dataset 61
write_json 63
write_ndjson 64
write_parquet 66
write_rds 68
write_spec 69
write_template 72
write_xpt 74
xpt_members 76

Index **78**

`apply_spec` *Conform a data frame to its spec*

Description

Run the ordered, transactional artoo pipeline that turns a raw analysis data frame into one conformed to its specification and carrying `artoo_meta`. This is the middle of the workflow (spec -> apply_spec -> read_/write_): the conformed frame is ready for any `write_*()` codec, and the metadata it now carries makes that write lossless. The input is never mutated; if any step aborts, the call leaves your data untouched.

Usage

```
apply_spec(
  x,
  spec,
  dataset,
  conformance = c("warn", "abort", "off"),
  na_position = c("first", "last"),
  extra = c("keep", "drop"),
```

```
  on_coercion_loss = c("error", "keep")
)
```

Arguments

x	*The raw data frame to conform.* <data.frame>: required.
spec	*The specification to conform to.* <artoo_spec>: required.
dataset	*The dataset whose rules apply.* <character(1)>: required. Must name a dataset in spec.
conformance	*What to do with conformance findings.* <character(1)>. One of: • "warn" (default) run check_spec(), attach the findings (read them with conformance()), warn on any error-severity finding. • "abort" abort with artoo_error_conformance on any error-severity finding. • "off" skip the check entirely. **Note:** this governs only the *findings* disposition — what is *reported*. Pipeline errors are a different category and abort under every setting, including "off": an unknown dataset, and lossy coercion (artoo_error_type). Lossy coercion has its own governed gate, on_coercion_loss, not this argument; when the spec is the problem, retype it with set_type().
na_position	*Where missing key values sort.* <character(1)>. One of "first" (default) or "last". "first" matches SAS PROC SORT (and the FDA submission convention) by ordering missings before present values; "last" matches R's order() and the pandas/Polars default. Both are lossless; pick the one your comparison target uses.
extra	*What happens to undeclared columns.* <character(1)>. An "extra" is a column of x the spec does not declare — typically a derivation temporary. One of: • "keep" (default) extras ride along after the declared columns, reported by the extra_variable finding. • "drop" the returned frame carries exactly the spec's columns; the drop is announced (artoo_message_apply), and that message is the audit trail of what was removed. **Interaction:** the drop runs *before* the check, so conformance() reports only the columns the returned frame keeps. Under conformance = "abort" an error-severity finding still aborts (those findings arise only on spec-declared columns, which the drop never touches) and the input is never mutated, so the trim cannot mask a failure. **Note:** "keep" is the default deliberately. artoo is a lossless carrier, so the metadata step never silently discards a column; extras are surfaced every run (the extra_variable finding, and a warning under conformance = "warn"), making "drop" a conscious opt-in.
on_coercion_loss	*What to do when coercion would lose data.* <character(1)>. The governed gate for an integer dataType whose data truncates (fractions) or overflows (R's 32-bit range). One of:

- `"error"` (default) abort with `artoo_error_type` before any value is touched, refusing to damage the data.
- `"keep"` skip coercion for the offending column, leaving it at its wider source type. The values are preserved and the mismatch is reported as an `integer_fraction` / `integer_overflow` finding (read with `conformance()`), never silently truncated.

Interaction: independent of `conformance`. `"error"` aborts even under `conformance = "off"`; under `"keep"` the finding it leaves is surfaced by `conformance = "warn"` (the default) and suppressed by `"off"`.

Tip: `"keep"` is the iterate stance (preserve the data, flag the spec); `"error"` is the submission stance. To fix the spec itself, see `set_type()` and `repair_spec()`.

Details

Ordered pipeline. Four fixed steps run in order: coerce each column to its CDISC dataType, reorder columns to the spec, sort rows by the dataset keys, then stamp the metadata. A spec variable the data lacks is never fabricated as an empty column: artoo is a lossless carrier, not a deriver. It is reported instead, an informational heads-up at apply time plus a `missing_variable` finding (when mandatory) or `missing_permissible` (when not), and left absent, so the conformed frame carries only the columns the data actually had.

Extras are kept by default. A column the spec does not declare survives the pipeline (ordered after the declared ones), is *reported* by the `extra_variable` conformance finding, and round-trips through every `write_*()` codec with metadata inferred from its R class — membership reported, never enforced by silent destruction. Keeping is the default because artoo is lossless by construction: a metadata-application step that silently discarded columns would break that contract, so trimming data is always an explicit, announced choice rather than a default side effect. `extra = "drop"` opts in to trim-to-spec (the returned frame carries exactly the spec's columns): the undeclared columns are removed *before* the check, so the findings describe exactly the returned frame (a dropped column is never reported as `extra_variable`), and the drop itself is always announced (`artoo_message_apply`) as the audit trail of what was removed — even under `conformance = "off"`.

Lossless or abort, your call. A coercion that would damage values — an `integer` dataType truncating fractions or overflowing R's 32-bit range — aborts with `artoo_error_type` before any value is touched, under the default `on_coercion_loss = "error"`. This gate is independent of `conformance`: `conformance = "off"` does not bypass it. When the data (not the spec) is right, set `on_coercion_loss = "keep"`: the column keeps its wider source type and the divergence is reported as an `integer_fraction` / `integer_overflow` finding, never silently truncated. When the spec is wrong, retype it with `set_type()` (or `repair_spec()` from the findings). The error abort carries the offending rows as data: `cnd$variables` is a data frame with columns `variable`, `data_type`, `n`, and `reason` (`"truncated"` / `"overflowed"`), so a pipeline can collect every mismatch in one `tryCatch(..., artoo_error_type = function(cnd) cnd$variables)` pass. The NA-introduction warning (`artoo_warning_coercion`) carries the same frame with `reason = "na_introduced"`, and a `conformance = "abort"` failure carries the complete findings frame as `cnd$findings`.

Values are never translated. Coded variables keep their submission values (`SEX` stays `"M"`); codelist translation is its own verb, `decode_column()`.

Value

A conformed <data.frame> carrying artoo_meta (read it with get_meta()) and, unless conformance = "off", the findings frame conformance() reads back. Hand it to any write_*() codec.

See Also

Check: check_spec() for the findings; conformance() to read them back.

Fix the spec: set_type() to retype a variable the data disagrees with, repair_spec() to apply every integer fix from a findings frame.

Translate: decode_column() for codelist value mapping.

Metadata: get_meta() / set_meta() for what the stamp attaches.

Examples

```
# ---- Example 1: conform ADSL, then read its metadata ----
#
# The bundled adam_spec describes ADSL; the raw frame is coerced,
# ordered, sorted, and stamped with the CDISC metadata get_meta() reads
# back. Variables the spec declares but this extract never derived are
# reported (not added), readable via conformance().
adsl <- apply_spec(cdisc_adsl, adam_spec, "ADSL")
get_meta(adsl)@dataset$records

# ---- Example 2: extras are kept and reported, or dropped on request ----
#
# By default a column outside the spec rides along (reported by the
# extra_variable finding) and still writes losslessly; extra = "drop"
# trims to the spec, announced and still reported. DM is SDTM, so it
# conforms against the bundled sdtm_spec.
raw <- cdisc_dm
raw$DERIVED <- seq_len(nrow(raw))
dm <- apply_spec(raw, sdtm_spec, "DM")
findings <- conformance(dm)
findings[findings$check == "extra_variable", c("variable", "message")]
trimmed <- apply_spec(raw, sdtm_spec, "DM", extra = "drop")
"DERIVED" %in% names(trimmed)
```

---

artoo_checks *Control which conformance checks run*

Description

Build a reusable control that selects which dimensions check_spec() evaluates. Construct one per study and thread it through every check_spec() call so the conformance surface is consistent. Each toggle is validated at construction, so a mistyped name or value aborts early rather than being silently ignored.

Usage

```
artoo_checks(
  missing_variable = TRUE,
  missing_permissible = TRUE,
  extra_variable = TRUE,
  type_mismatch = TRUE,
  invalid_encoding = TRUE,
  length_overflow = TRUE,
  char_length_limit = TRUE,
  codelist_membership = TRUE,
  codelist_membership_extensible = TRUE,
  label_match = TRUE,
  key_uniqueness = TRUE,
  display_format = TRUE,
  variable_name = TRUE,
  dataset_name = TRUE,
  label_length = TRUE,
  integer_overflow = TRUE,
  integer_fraction = TRUE,
  iso8601_format = TRUE
)
```

Arguments

`missing_variable`
: *Flag mandatory spec variables absent from the data.* `<logical(1)>: default TRUE`.

`missing_permissible`
: *Flag permissible (non-mandatory) spec variables absent from the data.* `<logical(1)>: default TRUE`.

`extra_variable`
: *Flag data columns the spec does not declare.* `<logical(1)>: default TRUE`.

`type_mismatch`
: *Flag columns whose storage differs from the spec dataType.* `<logical(1)>: default TRUE`.

`invalid_encoding`
: *Flag character values whose bytes are not valid UTF-8.* `<logical(1)>: default TRUE`. Invalid bytes mean the source was read under a mis-declared encoding; re-read it with the true `encoding=` before the corruption reaches a writer.

`length_overflow`
: *Flag character values longer than the spec length.* `<logical(1)>: default TRUE`.

`char_length_limit`
: *Flag character values longer than the SAS XPORT v5 / FDA 200-byte limit.* `<logical(1)>: default TRUE`.

`codelist_membership`
: *Flag values outside their closed codelist.* `<logical(1)>: default TRUE`.

`codelist_membership_extensible`
: *Flag values outside an extensible codelist's enumerated terms.* `<logical(1)>: default TRUE`. A codelist whose `extended` flag is `TRUE` allows sponsor terms, so a non-member is a note, never an error; this toggle silences those notes independently of `codelist_membership`.

`label_match`
: *Flag a column whose label attribute differs from the spec label.* `<logical(1)>: default TRUE`.

`key_uniqueness` *Flag a dataset whose spec key variables do not uniquely identify its rows.* `<logical(1)>: default TRUE`.

`display_format` *Flag a date/datetime/time variable whose displayFormat is not a recognized SAS format of that family.* `<logical(1)>: default TRUE`.

`variable_name` *Flag a data column name that violates the XPORT naming rules.* `<logical(1)>: default TRUE`. Over 8 characters (the v5 limit), over 32 (the v8 limit), or containing anything but ASCII letters, digits, and underscore.

`dataset_name` *Flag a dataset name that violates the XPORT naming rules.* `<logical(1)>: default TRUE`. Same limits as `variable_name`.

`label_length` *Flag a column label attribute over the 40-byte XPORT v5 / FDA limit.* `<logical(1)>: default TRUE`.

`integer_overflow`
Flag an integer-typed variable holding values beyond R's 32-bit integer range. `<logical(1)>: default TRUE`. Such values become `NA` under coercion, so this is an error, not a warning.

`integer_fraction`
Flag an integer-typed variable holding fractional values. `<logical(1)>: default TRUE`. Coercion would truncate them (162.6 becomes 162) — a data-integrity event; fix the spec dataType (`float` / `decimal`) or the data before conforming.

`iso8601_format` *Flag a character date/datetime/time variable whose values are not valid ISO 8601 text.* `<logical(1)>: default TRUE`. A character column under a temporal dataType is the CDISC `--DTC` form; complete values, right-truncated partials (`"1951"`, `"1951-12"`), and SDTMIG hyphen placeholders (`"2003---15"`) all pass, while `"12NOV2019"` or an impossible calendar date is flagged.

Details

Selection, not severity. This control decides which findings are *produced*; `apply_spec()`'s `conformance` argument (warn, abort, off) decides what to *do* with the findings its full-default check raises. A disabled dimension is skipped entirely, so the findings frame stays clean.

Value

A `<artoo_checks>` *control object*. Pass it as the `checks` argument to `check_spec()`.

See Also

`check_spec()`, which consumes it; `apply_spec()` for the findings disposition.

Examples

```
# ---- Example 1: the default runs every conformance dimension ----
#
# With no arguments, every conformance dimension is enabled.
artoo_checks()

# ---- Example 2: silence one dimension for a whole study ----
#
# Turn off the length check (e.g. while a spec's lengths are provisional)
# and reuse the control across every dataset.
```

```
spec <- artoo_spec(cdisc_sdtm_datasets, cdisc_sdtm_variables, codelists = cdisc_codelists)
ck <- artoo_checks(length_overflow = FALSE)
nrow(check_spec(cdisc_dm, spec, "DM", checks = ck))
```

`artoo_encodings` *Encodings for clinical datasets, across R, SAS, and Python*

Description

List the character encodings clinical data actually travels in, with the name each ecosystem uses for the same thing: the R name (the standard IANA name, which `iconv()` and the wider R ecosystem use), the SAS session-encoding name, and the Python codec. Any spelling from the `r` or `sas` column works as the `encoding` argument of every artoo reader and writer.

Usage

```
artoo_encodings()
```

Details

What an encoding is. Text is stored as bytes; an encoding is the rule that maps those bytes to characters. Plain A-Z digits and punctuation are the same bytes in every encoding listed here — the differences only show in accented letters (a-umlaut, e-acute), special symbols (micro, degree), and non-Latin scripts. Reading bytes with the wrong rule is what turns a degree sign into garbage.

Which one do I have? In SAS, run `PROC OPTIONS OPTION=ENCODING; RUN;` and look up the reported name in the `sas` column. Most US/EU Windows SAS installs report `WLATIN1` — that is `windows-1252` here.

Which one should I write? Usually none: `write_*(encoding = NULL)` (the default) inherits the encoding recorded when the data was read, so a round-trip is byte-faithful. The regulatory defaults artoo applies when nothing is recorded: SAS XPORT writes `US-ASCII` (the FDA Study Data Technical Conformance Guide expectation) and Dataset-JSON / NDJSON write `UTF-8` (required by CDISC and RFC 8259). A value that cannot be represented in the target encoding aborts loudly — see `on_invalid` on the writers.

Note: in memory, artoo text is always UTF-8 (NFC-normalised) — encodings only matter at the file boundary, exactly as in Python 3.

Value

A `<data.frame>` with one row per encoding and columns `r` (the R name — the standard IANA name `iconv()` uses, and what artoo records in the metadata), `sas` (the SAS session-encoding name), `python` (the Python codec name), and `description`.

See Also

Use it: the `encoding` argument of `read_xpt()`, `write_xpt()`, `read_json()`, and the other readers/writers.

Formats: `artoo_formats()` for the codec registry.

Examples

```
# ---- Example 1: the full cross-ecosystem table ----
#
# One row per encoding; the same byte rule under each ecosystem's name.
artoo_encodings()

# ---- Example 2: look up a SAS session encoding ----
#
# PROC OPTIONS reported WLATIN1: find the R and Python names for the
# same bytes (the sas and r spellings both work as encoding=).
enc <- artoo_encodings()
enc[enc$sas == "WLATIN1", ]
```

---

artoo_formats *Report which formats are available*

Description

List every registered codec and whether it can read and write in this session. Every format artoo ships is available: the pure-R codecs need nothing, and the one engine-backed codec (parquet) names a package artoo depends on rather than suggests. The columns exist because the registry supports an engine that may be absent, not because one currently is. Purely informational, modelled on the diagnostic helpers in the wider ecosystem; it never aborts.

Usage

```
artoo_formats()
```

Value

A `<data.frame>` with one row per format and columns `format`, `read`, `write` (logical), and `extensions`.

See Also

`read_dataset()` and `write_dataset()` which use the registry.

Examples

```
# ---- Example 1: see what this session can read and write ----
#
# rds is always available; the table shows the extensions each codec claims.
artoo_formats()
```

`artoo_spec` *Construct a CDISC specification*

Description

Build and validate a `artoo_spec` from dataset, variable, and codelist tables. Each table is coerced to a plain data frame, missing optional columns are filled with typed `NA`s, every variable type is canonicalised to the CDISC `dataType` vocabulary, and cross-slot integrity (dataset and codelist references) is checked before the object is returned. The spec is the lingua franca the rest of artoo reads, applies, and serialises.

Usage

```
artoo_spec(
  datasets = NULL,
  variables = NULL,
  codelists = NULL,
  study = NULL,
  values = NULL,
  methods = NULL,
  comments = NULL,
  documents = NULL,
  standard = NULL,
  standards = NULL,
  where_clauses = NULL,
  method_expressions = NULL,
  arm_displays = NULL,
  arm_results = NULL,
  dictionaries = NULL
)
```

Arguments

`datasets`	*Dataset-level metadata table.* `<data.frame>: required`. One row per dataset; must carry a `dataset` column. Optional columns `label`, `class`, `structure`, `keys` are filled with `NA` when absent.
`variables`	*Variable-level metadata table.* `<data.frame>: required`. One row per variable; must carry `dataset`, `variable`, and `data_type`. The `data_type` column is canonicalised to a CDISC `dataType` (e.g. `"text"` becomes `"string"`). **Requirement:** every `dataset` value must appear in `datasets`.
`codelists`	*Controlled-terminology terms.* `<data.frame> | NULL`. Must carry `codelist_id` and `term` when supplied. **Interaction:** every `codelist_id` referenced by `variables` must resolve here.
`study`	*Study-level metadata.* `<data.frame> | NULL`. A single row of named study fields. Well-known fields are canonicalised to `study_name`, `study_description`,

and protocol_name (aliases such as StudyName or studyid resolve automatically); other fields pass through verbatim. A standard field, when present, is consumed into @standard.

values *Value-level (VLM) metadata.* <data.frame> | NULL.

methods *Derivation methods.* <data.frame> | NULL. The Define-XML method definitions variables reference by method_id; must carry method_id when supplied. Completeness (e.g. a referenced method has a description) is checked by validate_spec(), not here.

comments *Comment definitions.* <data.frame> | NULL. Referenced by comment_id; must carry comment_id when supplied.

documents *Document references.* <data.frame> | NULL. Referenced by document_id; must carry document_id when supplied.

standard *The primary CDISC standard the spec implements.* <character(1)> | NULL. E.g. "ADaMIG 1.1" or "SDTMIG 3.2". When NULL (default) it is resolved from study$standard, or from the value most rows of datasets$standard name; absent everywhere, @standard is NA.

Restriction: an explicit value that matches nothing the source names aborts with artoo_error_spec.

standards *CDISC standards this spec claims.* <data.frame> | NULL. Must carry standard_id, name and version. Define-XML 2.1 emits these as a def:Standards block that datasets and codelists reference by id; 2.0 has room for only one, taken from the row flagged is_primary.

where_clauses *Structured value-level conditions.* <data.frame> | NULL. Must carry where_clause_id and comparator. One row per check value, because a check value is free text and may itself contain a comma, so any collapsed form would be lossy.

method_expressions

Formal expressions for derivation methods. <data.frame> | NULL. Must carry method_id. A separate table because a method may carry several expressions in different languages, which extra rows on methods could not express without changing what the published one-row-per-method rule means.

arm_displays *Analysis result displays.* <data.frame> | NULL. Must carry display_id. Analysis Results Metadata is version-neutral: the vocabulary is identical for Define-XML 2.0 and 2.1.

arm_results *Analysis results.* <data.frame> | NULL. Must carry display_id and result_id. One row per result and analysis dataset, since each analysis dataset carries its own where-clause reference.

dictionaries *External codelists.* <data.frame> | NULL. Must carry dictionary_id. A terminology too large to enumerate, named rather than listed: MedDRA, WHODrug, ISO 3166. Both readers populate it, and a variable points at one from the same codelist_id column it would use for an enumerated list.

Details

Coerce, then validate. Each table is first coerced to a plain data frame (a tibble is accepted and demoted); known columns are cast to their storage mode and absent optional columns are added as

typed NA, so every downstream reader can trust the schema. Validation runs only after coercion, on the completed slots.

Type canonicalisation. variables$data_type is mapped through the closed CDISC dataType vocabulary (string, integer, decimal, float, double, boolean, date, datetime, time, URI). Common SAS / P21 spellings resolve automatically ("text", "Char", "integer (8)", ...); an unrecognised token aborts with artoo_error_type.

Cross-slot integrity. Construction fails (artoo_error_spec) if a variable names a dataset absent from datasets, or references a codelist_id absent from codelists.

One primary standard, linked per dataset. The scalar @standard property holds the spec's primary CDISC standard, resolved from the standard argument, a standard column in datasets (the P21 workbook shape), and a standard field in study (the Define-XML shape) — those columns are consumed, so @standard is the single home. A datasets column naming several standards is legitimate (a study may mix implementation-guide versions): each row is linked to its standard via datasets$standard_id, minting a standards row where none defines the name, and @standard takes the study's stated standard, or failing that the one most datasets name. An explicit standard argument contradicting every value in the source aborts with artoo_error_spec.

One study vocabulary. Well-known study fields are canonicalised to the CDISC ODM GlobalVariables names, snake_cased: study_name, study_description, protocol_name. Source spellings resolve automatically (StudyName, studyid, ...); fields the vocabulary does not know pass through verbatim. Aliases that disagree on a value abort with artoo_error_spec.

Value

A validated artoo_spec *object.* Inspect it with spec_datasets() / spec_variables(), or check it with validate_spec().

See Also

Inspect: spec_datasets(), spec_variables(), spec_codelists(), spec_keys(), spec_study().

Check: validate_spec(). **Predicate:** is_artoo_spec().

Examples

```
# ---- Example 1: build a spec from the bundled CDISC-pilot tables ----
#
# `cdisc_sdtm_datasets` and `cdisc_sdtm_variables` hold the CDISC pilot SDTM
# metadata in the shape artoo_spec() expects; the constructor
# canonicalises every type and checks cross-slot integrity.
spec <- artoo_spec(cdisc_sdtm_datasets, cdisc_sdtm_variables, codelists = cdisc_codelists)
spec_datasets(spec)

# ---- Example 2: a focused spec for a single dataset ----
#
# Slice the bundled tables to one dataset (DM) to build a smaller spec.
dm_ds <- cdisc_sdtm_datasets[cdisc_sdtm_datasets$dataset == "DM", ]
dm_var <- cdisc_sdtm_variables[cdisc_sdtm_variables$dataset == "DM", ]
dm_spec <- artoo_spec(dm_ds, dm_var, codelists = cdisc_codelists)
head(spec_variables(dm_spec, "DM")[, c("variable", "label", "data_type")])
```

cdisc_adae *Demo adverse events analysis dataset (ADaM ADAE)*

Description

A 60-row sample of the CDISC pilot ADaM adverse events analysis dataset (ADAE): one row per reported event, with treatment-emergent flags, severity, and coding variables (labels preserved as attributes).

Usage

```
cdisc_adae
```

Format

A data frame with 60 rows (`STUDYID`, `USUBJID`, `AETERM`, `AESEV`, `TRTEMFL`, `ASTDT`, ...).

Source

First 60 rows of the CDISC pilot `adam/cdisc/adae.xpt` from the PHUSE Test Data Factory (`phuse-org/phuse-scripts`).

cdisc_adsl *Demo subject-level analysis dataset (ADaM ADSL)*

Description

A 60-subject sample of the CDISC pilot ADaM subject-level analysis dataset (ADSL): one row per subject, with treatment, demographic, baseline, and disposition variables (labels preserved as column attributes).

Usage

```
cdisc_adsl
```

Format

A data frame with 60 rows and 48 variables (`STUDYID`, `USUBJID`, `TRT01P`, `AGE`, `SEX`, `RACE`, `SAFFL`, `TRTSDT`, ...).

Source

First 60 subjects of the CDISC pilot `adam/cdisc/adsl.xpt` from the PHUSE Test Data Factory (`phuse-org/phuse-scripts`).

cdisc_dm *Demo demographics dataset (SDTM DM)*

Description

A 60-subject sample of the CDISC pilot SDTM demographics domain (DM): one row per subject, with the standard DM variables (labels preserved as attributes).

Usage

```
cdisc_dm
```

Format

A data frame with 60 rows and 25 variables (`STUDYID`, `DOMAIN`, `USUBJID`, `AGE`, `SEX`, `RACE`, `ARM`, `COUNTRY`, ...).

Source

First 60 subjects of the CDISC pilot `sdtm/TDF_SDTM_v1.0/dm.xpt` from the PHUSE Test Data Factory (`phuse-org/phuse-scripts`).

cdisc_spec *CDISC demo specification tables (one standard per pair)*

Description

The constructor-shaped metadata tables for the bundled demo data, split by CDISC standard because a `artoo_spec` carries exactly one: `cdisc_adam_datasets` + `cdisc_adam_variables` describe ADSL (ADaMIG 1.1), and `cdisc_sdtm_datasets` + `cdisc_sdtm_variables` describe DM (SDTMIG 3.1.2). Each variables table is *derived from the data* (names, labels, inferred CDISC types, byte lengths) by `data-raw/`. Pass one standard's pair to `artoo_spec()`; passing both pairs together aborts with `artoo_error_spec` — mixing standards in one spec is the mistake the split exists to prevent.

Usage

```
cdisc_adam_datasets

cdisc_adam_variables

cdisc_sdtm_datasets

cdisc_sdtm_variables

cdisc_codelists
```

Format

Each `*_datasets` table is a data frame with one row per dataset:

dataset Dataset name (`"ADSL"` or `"DM"`).

label Dataset label.

standard The CDISC standard, consumed into the spec's `spec_standard()`.

Each `*_variables` table is a data frame with one row per variable:

dataset Owning dataset name.

variable Variable name.

label Variable label (from the data's `label` attribute).

data_type CDISC `dataType` inferred from the column's class.

length Storage length (max byte width for character, 8 for numeric).

order Variable order within the dataset.

codelist_id NCI codelist reference (`"C66731"` on `SEX`).

`cdisc_codelists` is a data frame of controlled-terminology terms (the real NCI codelist C66731 for `SEX`):

codelist_id Codelist identifier (`"C66731"`).

term Submission value (`"M"`, `"F"`, ...).

decode Decoded value (`"Male"`, `"Female"`, ...).

order Term order.

Source

Derived from the CDISC pilot `.xpt` files in the public PHUSE Test Data Factory (`phuse-org/phuse-scripts`) by `data-raw/bundle-demo.R`.

`cdisc_specs` *Bundled CDISC specifications (ADaM and SDTM)*

Description

Ready-made `artoo_spec` objects built from the official CDISC Define-XML 2.1 release examples: `adam_spec` (ADaMIG 1.1; datasets ADSL, ADAE) and `sdtm_spec` (SDTMIG 3.1.2; datasets TS, DM, VS, SUPPDM). Every bundled demo dataset conforms to its spec under `apply_spec(conformance = "abort")` — the pairing is gated at build time. The same specs ship as P21 workbooks in `system.file("extdata", "adam-spec.xlsx", package = "artoo")` and `"sdtm-spec.xlsx"`, written by `write_spec()`.

Usage

```
adam_spec

sdtm_spec
```

Format

A validated `artoo_spec()` object; inspect it with `spec_datasets()`, `spec_variables()`, and `spec_standard()`.

Details

Demo adaptations (each an ADR in `data-raw/bundle-spec.R`): the sponsor-defined codelists (`CL.ARM`, `CL.ARMCD`, `CL.BMICAT`, and the extensible NCI VS codelists) are marked `extended`; `VISITNUM` is typed `float` (the pilot data has fractional visit numbers); VS declares the SDTMIG timepoint variables `VSTPT`/`VSTPTNUM`, with `VSTPTNUM` in the VS key.

Source

The CDISC Define-XML 2.1 release example defines (ADaM + SDTM), pinned by sha256 in `data-raw/bundle-spec.R`; data from the PHUSE Test Data Factory.

---

cdisc_suppdm *Demo supplemental qualifiers dataset (SDTM SUPPDM)*

Description

A 60-row sample of the CDISC pilot SDTM supplemental qualifiers for DM (SUPPDM): the non-standard qualifier values that ride alongside the DM domain.

Usage

```
cdisc_suppdm
```

Format

A data frame with 60 rows (`STUDYID`, `RDOMAIN`, `USUBJID`, `QNAM`, `QVAL`, ...).

Source

First 60 rows of the CDISC pilot `sdtm/cdiscpilot01/suppdm.xpt` from the PHUSE Test Data Factory (`phuse-org/phuse-scripts`).

cdisc_ts *Demo trial summary dataset (SDTM TS)*

Description

The CDISC pilot SDTM trial summary domain (TS): one row per trial characteristic (33 rows in the pilot), the study-design parameters a submission carries.

Usage

```
cdisc_ts
```

Format

A data frame with 33 rows (`STUDYID`, `TSPARMCD`, `TSPARM`, `TSVAL`, ...).

Source

The CDISC pilot `sdtm/cdiscpilot01/ts.xpt` from the PHUSE Test Data Factory (`phuse-org/phuse-scripts`).

cdisc_vs *Demo vital signs dataset (SDTM VS)*

Description

A 60-row sample of the CDISC pilot SDTM vital signs domain (VS): repeated measurements per subject across visits, positions, and planned timepoints.

Usage

```
cdisc_vs
```

Format

A data frame with 60 rows (`STUDYID`, `USUBJID`, `VSTESTCD`, `VSORRES`, `VISITNUM`, `VSPOS`, `VSTPTNUM`, ...).

Source

First 60 rows of the CDISC pilot `sdtm/cdiscpilot01/vs.xpt` from the PHUSE Test Data Factory (`phuse-org/phuse-scripts`).

check_spec *Check a dataset against its spec*

Description

Compare a data frame to one dataset's specification and report where they diverge. This is the data-conformance check at the end of the artoo workflow (spec -> apply_spec -> check_spec): it reuses the metadata the spec already carries (variables, types, lengths, codelists, keys). It is distinct from `validate_spec()`, which checks the spec's own internal integrity rather than the data. Both report findings keyed to the same open rule catalog.

Usage

```
check_spec(
  x,
  spec,
  dataset,
  decode = c("none", "to_decode", "to_code"),
  checks = NULL
)
```

Arguments

x	*The data frame to check.* `<data.frame>: required`. Typically the output of `apply_spec()`, but any frame works.
spec	*The specification to check against.* `<artoo_spec>: required`.
dataset	*The dataset whose rules apply.* `<character(1)>: required`. **Restriction:** must name a dataset in spec (see `spec_datasets()`).
decode	*Which codelist column membership is checked against.* `<character(1)>`. One of `"none"` (default), `"to_decode"`, or `"to_code"`.
checks	*Which conformance dimensions to evaluate.* `<artoo_checks> | NULL`. When `NULL` (default) every dimension runs; build a control with `artoo_checks()` to disable some.

Details

Findings, not enforcement. `check_spec()` never modifies data; it returns every divergence it finds. `apply_spec()` runs it and decides what to do via its `conformance` argument (warn, abort, off). The dimensions checked are: missing variables (split into mandatory, an error, and permissible, a warning), extra variables (data column the spec does not declare), type mismatch, ISO 8601 validity of character date/datetime/time values (CDISC partials pass; `"12NOV2019"` does not), fractional values and 32-bit overflow under an `integer` dataType (both would corrupt data at coercion), character length overflow, the hard 200-byte XPORT v5 / FDA character limit, codelist membership, label drift against the spec, key uniqueness, and displayFormat validity.

Decode-aware membership. `decode` selects which codelist column the data is checked against, matching `apply_spec()`'s decode step: `"none"`/`"to_code"` check against the codelist `terms`, `"to_decode"`

against the decodes. apply_spec() threads its own decode through, so a decoded column is not wrongly flagged.

Fatal vs informational coercion checks. Only integer_fraction and integer_overflow carry error severity: they mark data an integer dataType cannot hold without loss, which apply_spec() refuses to coerce (its on_coercion_loss governs that gate). type_mismatch is a note: a column stored more widely than the spec declares (an integer-valued double, for instance) coerces cleanly, so it is informational, not a blocker.

Value

A findings data frame with columns check, dimension, severity ("error", "warning", or "note"), dataset, variable, and message, one row per divergence. Zero rows means the data conforms.

See Also

apply_spec() which runs this; check_study() for the same check across a whole study; artoo_checks() to select dimensions; validate_spec() for spec integrity.

Examples

```
# ---- Example 1: a conformed frame surfaces only the genuine gaps ----
#
# apply_spec() coerces and orders to spec but never fabricates a variable
# the data lacks; checking the result reports the permissible variables
# this extract never derived (here, six) instead of hiding them as empty
# columns.
adsl <- apply_spec(cdisc_adsl, adam_spec, "ADSL", conformance = "off")
nrow(check_spec(adsl, adam_spec, "ADSL"))

# ---- Example 2: raw data surfaces divergences ----
#
# Checking a raw frame with an undeclared column flags the extras.
raw <- cdisc_adsl
raw$NOTASPEC <- 1
head(check_spec(raw, adam_spec, "ADSL")[, c("check", "variable", "severity")])
```

---

check_study *Check a whole study against its spec*

Description

Run check_spec() over every dataset in a study and return one stacked findings frame. Where check_spec() answers "does this dataset conform?", check_study() answers "is my whole study submittable?" in a single pass, surfacing every dataset's divergences at once instead of one abort at a time. The result is an ordinary findings frame underneath, so filter it by severity or hand it straight to repair_spec().

Usage

```
check_study(
  spec,
  data,
  decode = c("none", "to_decode", "to_code"),
  checks = NULL
)
```

Arguments

spec	*The specification to check against.* <artoo_spec>: required.
data	*The study's datasets.* <named list of data.frame>: required. One entry per dataset, named by the dataset (e.g. list(ADSL = adsl, ADAE = adae)). Every name must be a dataset in spec.
decode	*Which codelist column to check against.* <character(1)>. Passed to check_spec(); one of "none" (default), "to_decode", "to_code".
checks	*Which conformance dimensions to run.* <artoo_checks> | NULL. Passed to check_spec(); NULL (default) runs every dimension. Build a subset with artoo_checks().

Details

One row per divergence, every dataset stacked. Each dataset's findings carry its name in the dataset column, so the frame is the union of the per-dataset check_spec() results. Printing renders the dataset-by-check count matrix (the study-level summary); the underlying frame is unchanged.

Data-requiring, like check_spec(). check_study() checks data against the spec, so it needs the data frames. For the spec's own structural integrity (no data), use validate_spec().

Value

A <artoo_study_findings> *data frame* with the same columns as check_spec() (check, dimension, severity, dataset, variable, message), one row per divergence across all datasets. Zero rows means the whole study conforms. Print it for the count matrix; treat it as an ordinary data frame otherwise.

See Also

One dataset: check_spec(). **Spec structure only:** validate_spec().

Repair: repair_spec() to apply the integer fixes the matrix surfaces.

Examples

```
# ---- Example 1: scan a whole study in one pass ----
#
# Loop the conformance check over every dataset's data. A fractional AGE
# (the spec types it integer) surfaces as an integer_fraction finding; the
# print is a dataset-by-check count matrix.
```

```
adsl <- cdisc_adsl
adsl$AGE <- adsl$AGE + 0.5
check_study(adam_spec, list(ADSL = adsl, ADAE = cdisc_adae))

# ---- Example 2: feed the findings straight into repair_spec() ----
#
# The result is an ordinary findings frame, so repair_spec() consumes it to
# flip every integer_fraction / integer_overflow variable across the study.
findings <- check_study(adam_spec, list(ADSL = adsl))
fixed <- repair_spec(adam_spec, findings)
spec_variables(fixed, "ADSL")$data_type[
  spec_variables(fixed, "ADSL")$variable == "AGE"
]
```

`columns` *View a dataset's variable attributes, SAS-style*

Description

Return a one-row-per-variable attribute table — the pane a SAS programmer reads in `PROC CONTENTS` or the Universal Viewer: position, name, Char/Num type, length, format, informat, label, and the CDISC key sequence. This is the quick look after `apply_spec()` stamps a frame, or on any dataset file artoo can read.

Usage

```
columns(x, member = NULL)
```

Arguments

`x`	*What to describe.* `<data.frame> | <character(1)>: required`. A stamped frame (carries `artoo_meta`), any plain data frame, or a path to a dataset file (`.xpt`, `.json`, `.ndjson`, `.parquet`, `.rds`).
`member`	*XPORT member to describe.* `<character(1)> | NULL`. Only meaningful when `x` is a path to a multi-member `.xpt` file.

Details

Every real column shows. The table covers the *frame's* columns: a column the spec never declared (which `apply_spec()` keeps, never drops) still appears, its attributes inferred from the R class. A plain, never-stamped data frame works the same way — every attribute is inferred.

Len is physical storage. The pane mirrors what `PROC CONTENTS` shows and what the writers store, not a spec digit-width. A Char column always carries a byte `Len` (the declared length, else inferred from the data); a numeric `Len` is blank, because a numeric stores as an 8-byte IEEE double with no character width (a Define-XML numeric `Length` is a digit-width, kept in the metadata for the Define / P21 surface). Format and informat names render uppercase; the metadata keeps the source spelling.

A path reads through the codec. A file path is dispatched by extension through the same registry as `read_dataset()`, so the attributes come from the one lossless reader (an unknown extension aborts with the registry's known-extensions message).

Tip: a multi-member XPORT file needs `member =`; without one the xpt reader aborts and points at `xpt_members()` for the listing.

Note: an `.xpt` path shows a blank `Key`: the XPORT byte layout stores only name, label, length, and formats, so `keySequence` (like codelist and origin) cannot ride in the file. The metadata-carrying formats (`.json`, `.ndjson`, `.parquet`, `.rds`) and the in-session conformed frame show it; re-apply the spec after an xpt read to restore it.

Value

A `<artoo_columns>` *data frame* with columns `#`, `Variable`, `Type`, `Len`, `Format`, `Label`, `Key`, printed left-aligned. The `Informat` column appears only when at least one variable carries an informat (most clinical panes have none). It is an ordinary data frame underneath — filter or inspect it like one.

See Also

Members: `xpt_members()` lists a multi-member XPORT file.

Metadata: `get_meta()` for the full `artoo_meta`; `apply_spec()` which stamps it.

Examples

```
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)

# ---- Example 1: the column pane of a conformed frame ----
#
# apply_spec() stamps ADSL with its metadata; columns() reads it back as
# the SAS-style attribute table.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
columns(adsl)

# ---- Example 2: straight off a file ----
#
# Write the conformed frame to any format and point columns() at the
# path; the codec reads it back and the attributes are identical.
p <- tempfile(fileext = ".json")
write_json(adsl, p)
columns(p)
```

`conformance` *Read the conformance findings a dataset carries*

Description

Pull the conformance findings apply_spec() attached to a conformed data frame — the readable answer to "what did the check find?". The result is the same findings frame check_spec() returns (one row per divergence), with a print method that renders a sectioned report, so conformance(adsl) at the console is the inspection step the artoo_warning_conformance warning points you at.

Usage

```
conformance(x)
```

Arguments

x	*A data frame produced by* apply_spec(). <data.frame>: required. **Requirement:** the conformance check must have run: a frame from apply_spec(..., conformance = "off") (or one rebuilt by a transform that dropped attributes) carries no findings and aborts with artoo_error_input.

Value

A <artoo_findings> *data frame* with columns check, dimension, severity ("error", "warning", or "note"), dataset, variable, and message. Zero rows means the data conformed. Print it for the sectioned report; treat it as an ordinary data frame for programmatic use.

See Also

apply_spec() which attaches the findings; check_spec() for the same check on demand; artoo_checks() to select dimensions.

Examples

```
spec <- artoo_spec(
  cdisc_sdtm_datasets, cdisc_sdtm_variables,
  codelists = cdisc_codelists
)

# ---- Example 1: inspect what the conform step found ----
#
# Conforming raw DM records the findings on the result; conformance()
# renders them as a report instead of a raw attribute.
dm <- suppressWarnings(apply_spec(cdisc_dm, spec, "DM"))
conformance(dm)

# ---- Example 2: gate a pipeline on error-severity findings ----
#
# The findings frame is an ordinary data frame: filter by severity to
# drive your own logic.
f <- conformance(dm)
nrow(f[f$severity == "error", ])
```

decode_column *Derive or translate a variable through its codelist*

Description

Map one column's values through a spec codelist — code to decode or decode to code — writing the result to a new variable or in place. This is the everyday companion to `apply_spec()`'s whole-dataset decode step: deriving RACEN from RACE, recovering submission codes from decoded values, or decoding a single variable for display, without re-running the pipeline. When the target variable is declared in the spec, the result is also coerced to its dataType and labelled, so the new column lands conformed.

Usage

```
decode_column(
  x,
  spec,
  dataset,
  from,
  to = from,
  direction = c("to_decode", "to_code"),
  no_match = c("error", "keep", "na"),
  trim = TRUE,
  ignore_case = FALSE
)
```

Arguments

x	*The data frame to extend.* <data.frame>: required.
spec	*The specification carrying the codelists.* <artoo_spec>: required.
dataset	*The dataset whose variables apply.* <character(1)>: required. Must name a dataset in spec.
from	*The source column.* <character(1)>: required. Must be a column of x.
to	*The destination variable.* <character(1)>: default from". Defaults to translating in place. A to declared in the spec gets its dataType coercion and label; an undeclared 'to' is plain character.
direction	*Which way to map.* <character(1)>. One of: • "to_decode" (default) map codes to their decoded values ("M" becomes "Male"). • "to_code" map decoded values to their submission codes — the RACEN-from-RACE derivation.
no_match	*Policy for values absent from the codelist.* <character(1)>. One of "error" (default; abort with artoo_error_codelist, naming the unmatched values and the codelist), "keep" (carry the source value through), or "na".

`trim`	*Match after trimming whitespace.* `<logical(1)>: default TRUE`.
`ignore_case`	*Match case-insensitively.* `<logical(1)>: default FALSE`. Case differences are usually genuine CT violations, so this is opt-in.

Details

Which codelist applies. The codelist attached to `to` in the spec wins (the natural direction for RACEN-style derivations, where the numeric variable owns the code/decode pairs); when `to` declares none, `from`'s codelist is used. If neither variable references a codelist the call aborts — there is nothing to map through.

Mismatched surfaces chain. A single call maps through ONE codelist, so the winning codelist's terms (or decodes) must line up with the `from` values — the CDISC `*N` convention guarantees this for RACEN-style pairs, whose decodes are the character variable's submission values. When the two codelists share no value surface (say SEXN's decodes are `"Female"/"Male"` but SEX holds `"F"/"M"`), the unmatched values hit the `no_match` policy; translate in two hops instead — decode through `from`'s codelist first, then `to_code` through the destination's:

```
dm |>
  decode_column(spec, "DM", from = "SEX", to = "SEXDECD") |>
  decode_column(spec, "DM", from = "SEXDECD", to = "SEXN",
                direction = "to_code")
```

Soft matches are reported, never silent. Values that match only after trimming whitespace (or case-folding, when `ignore_case = TRUE`) still map, with a `artoo_warning_codelist` naming the variants — `check_spec()` always compares exactly, so clean the source for submission.

Value

The data frame `x` with the `to` column added (at the end) or replaced (in place), ready for the next pipeline step.

See Also

Whole-dataset decode: `apply_spec()` with `decode =`.

Inspect the terms: `spec_codelists()`. **Check membership:** `check_spec()`.

Examples

```
spec <- artoo_spec(cdisc_sdtm_datasets, cdisc_sdtm_variables, codelists = cdisc_codelists)

# ---- Example 1: decode a coded variable into a display column ----
#
# SEX is coded against C66731; map the codes to their decodes in a new
# column, leaving the submission values untouched.
dm <- decode_column(cdisc_dm, spec, "DM", from = "SEX", to = "SEXDECD")
table(dm$SEX, dm$SEXDECD)

# ---- Example 2: the RACEN pattern, a coded numeric from its decode ----
#
```

```
# Declare SEXN as an integer variable owning a numeric codelist, then
# derive it from SEX's decoded values: to_code maps each decode to its
# submission code, and the spec dataType makes the result integer.
vars <- rbind(
  cdisc_sdtm_variables,
  data.frame(
    dataset = "DM", variable = "SEXN", label = "Sex (N)",
    data_type = "integer", length = 8L, order = NA_integer_,
    codelist_id = "SEXN"
  )
)
cls <- rbind(
  cdisc_codelists,
  data.frame(
    codelist_id = "SEXN", term = c("1", "2"),
    decode = c("F", "M"), order = 1:2
  )
)
spec_n <- artoo_spec(cdisc_sdtm_datasets, vars, codelists = cls)
dm_n <- decode_column(cdisc_dm, spec_n, "DM",
  from = "SEX", to = "SEXN", direction = "to_code"
)
str(dm_n$SEXN)
```

`get_meta` *Read the metadata a dataset carries*

Description

Pull the `artoo_meta` off a data frame produced by `apply_spec()` or read back by any `read_*()` codec. The metadata travels as a single Dataset-JSON string in the frame's `metadata_json` attribute; `get_meta()` parses it to the S7 object, the form every codec writes from. This is the read half of the lossless round-trip.

Usage

```
get_meta(x)
```

Arguments

`x`	*A data frame carrying artoo metadata.* `<data.frame>: required`. Typically the output of `apply_spec()` or a `read_*()` codec. **Requirement:** `x` must carry a `metadata_json` attribute (set by `set_meta()`, `apply_spec()`, or a reader); a bare frame aborts with `artoo_error_input`.

Value

A <artoo_meta> with two properties. @dataset is a named list of dataset-level attributes: itemGroupOID, name, label, records, studyOID, metaDataVersionOID, encoding, and keys. @columns is a named list with one entry per variable, each carrying itemOID, name, label, dataType, targetDataType, length, displayFormat, informat, keySequence, codelist, significantDigits, and origin (absent values are NULL). Pass it to set_meta() to re-attach, or index it directly (meta@columnsAGElabel).

See Also

set_meta() for the write half; apply_spec() which stamps it.

Examples

```
# ---- Example 1: read metadata off a conformed dataset ----
#
# apply_spec() stamps the metadata; get_meta() reads it back as the S7
# object whose @columns holds one CDISC attribute set per variable.
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)
adsl <- apply_spec(cdisc_adsl, spec, "ADSL")
meta <- get_meta(adsl)
meta@columns$STUDYID

# ---- Example 2: round-trip metadata across two frames ----
#
# The metadata is a portable object: read it off one frame and stamp it
# onto another with set_meta().
bare <- as.data.frame(adsl)
attr(bare, "metadata_json") <- NULL
restamped <- set_meta(bare, meta)
identical(get_meta(restamped)@columns, meta@columns)
```

---

is_artoo_checks *Test for a artoo_checks control*

Description

Report whether an object is a artoo_checks control built by artoo_checks(). Use it to guard a checks argument before threading it into check_spec() or apply_spec().

Usage

```
is_artoo_checks(x)
```

Arguments

x	*Object to test.* <any>.

Value

A <logical(1)>: TRUE when x is a artoo_checks.

See Also

artoo_checks() to build one.

Examples

```
# ---- Example 1: confirm a control before reusing it ----
#
# is_artoo_checks() distinguishes a real control from a bare list of flags.
is_artoo_checks(artoo_checks())
is_artoo_checks(list(missing_variable = TRUE))
```

---

is_artoo_meta *Test for a artoo_meta object*

Description

Report whether an object is a artoo_meta — the CDISC-shaped metadata a conformed dataset carries through the artoo workflow (spec -> apply_spec -> read_/write_). get_meta() returns one; this is the type guard before you inspect its @dataset and @columns slots.

Usage

```
is_artoo_meta(x)
```

Arguments

x	*Object to test.* <any>.

Value

A <logical(1)>: TRUE when x is a artoo_meta, else FALSE.

See Also

get_meta() and set_meta() to read and attach metadata.

Examples

```
# ---- Example 1: guard before inspecting metadata ----
#
# get_meta() yields a artoo_meta; is_artoo_meta() confirms the type before
# you reach into its slots.
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)
adsl <- apply_spec(cdisc_adsl, spec, "ADSL")
meta <- get_meta(adsl)
is_artoo_meta(meta)

# ---- Example 2: a bare data frame carries no meta object ----
#
# The raw frame itself is not a artoo_meta — only the object get_meta()
# returns is.
is_artoo_meta(cdisc_adsl)
```

---

is_artoo_spec *Test for a artoo_spec object*

Description

Report whether an object is a `artoo_spec` — the validated CDISC specification that drives the artoo workflow (spec -> apply_spec -> read_/write_). `artoo_spec()` builds one; this is the type guard before you pass it to `apply_spec()` or reach into it with the spec accessors.

Usage

```
is_artoo_spec(x)
```

Arguments

x	*Object to test.* <any>.

Value

A <logical(1)>: `TRUE` when `x` is a `artoo_spec`, else `FALSE`.

See Also

`artoo_spec()` to build one; `is_artoo_meta()` for the metadata guard.

Examples

```
# ---- Example 1: guard a built specification ----
#
# artoo_spec() assembles and validates a spec; is_artoo_spec() confirms the
# type before you drive apply_spec() with it.
spec <- artoo_spec(cdisc_sdtm_datasets, cdisc_sdtm_variables, codelists = cdisc_codelists)
```

```
is_artoo_spec(spec)

# ---- Example 2: an ordinary object is not a spec ----
#
# Any non-artoo_spec value — a bare data frame, say — returns FALSE.
is_artoo_spec(cdisc_dm)
```

`lint_define` *Check the reference integrity of a Define-XML document*

Description

Reports OID references that resolve to nothing, and definitions that nothing references. Neither is expressible in XML Schema, so a document can pass `validate_define()` and still fail here.

Usage

```
lint_define(path)
```

Arguments

`path`	*Define-XML document to check.* `<character(1)>: required`.

Details

Findings are directional. A dangling reference and an orphan definition have different causes and different consequences, so they are separate conditions. Dangling references are errors. Orphans are warnings, except for an orphaned value list: a `def:ValueListDef` that no `def:ValueListRef` points at means every value-level definition it holds renders nowhere, which is a silent loss of submission metadata rather than untidiness.

Two exemptions prevent false positives. A `CodeList` backed by an `ExternalCodeList` (MedDRA, WHODrug, ISO 3166) is a dictionary reference and is not expected to be referenced by a `CodeListRef`. A `CodeList` reached through `ItemRef/@RoleCodeListOID` counts as referenced. Without these, every real adverse-event or medication define reports spurious orphans.

Value

An `artoo_check` *object.* Its `@findings` data frame has columns `check`, `dimension`, `severity`, `dataset`, `variable`, `message`, and is empty when every reference resolves and every definition is used.

See Also

Validate first: `validate_define()` for schema conformance, which this complements rather than repeats.

Examples

```
# ---- Example 1: a sound document ----
#
# The bundled minimal example resolves cleanly, so the findings table is
# empty and the summary counts what was inspected.
minimal <- system.file("extdata", "define-minimal.xml", package = "artoo")
report <- lint_define(minimal)
nrow(report@findings)
report@summary$n_definitions

# ---- Example 2: a reference that resolves to nothing ----
#
# Point a variable's codelist reference at an OID no CodeList defines. The
# document still passes schema validation; only the lint sees it.
broken <- tempfile(fileext = ".xml")
writeLines(
  sub('CodeListOID="CL.SEX"', 'CodeListOID="CL.MISSING"',
    readLines(minimal),
    fixed = TRUE
  ),
  broken
)
validate_define(broken)@summary$valid
lint_define(broken)@findings[, c("check", "severity", "message")]
```

`members` *List the datasets in a file or directory*

Description

Inventory the dataset(s) a path contains, one row per dataset, dispatched by extension through the same codec registry as `read_dataset()`. A SAS XPORT library lists every member; a single-dataset file (`.json`, `.ndjson`, `.parquet`, `.rds`) reports one row; a directory inventories each dataset file it holds. The format-neutral companion to the xpt-specific `xpt_members()`.

Usage

```
members(path, format = NULL)
```

Arguments

`path`	*A dataset file or a directory.* `<character(1)>: required`. A path to a dataset file (`.xpt`, `.json`, `.ndjson`, `.parquet`, `.rds`) or to a directory holding such files. A path that does not exist, or a file whose extension no codec claims, aborts.
`format`	*Restrict the inventory to these formats.* `<character> | NULL`. Defaults to `NULL`, which inventories every format. Format names as `artoo_formats()` lists them, not file extensions: `"parquet"` claims both `.parquet` and `.pq`. `NULL` inventories

every format. Several names are a set, not an order, so `c("xpt", "json")` lists both and says nothing about which wins.

Tip: the reason to pass it is a directory holding the same dataset in more than one format, where the full inventory lists `dm.xpt` and `dm.json` as two rows.

Restriction: it filters, it does not override. Unlike `read_dataset()`'s `format`, which reads a file AS the named format whatever its extension, this narrows which files are inventoried and leaves extension resolution alone. Naming one file whose format the restriction excludes aborts, rather than returning an empty inventory that could not be told apart from an empty directory.

Details

One dataset per file, except XPORT. XPORT is the only multi-dataset container artoo handles, so only an `.xpt` path can return more than one row. Every other format is one dataset per file.

A directory is inventoried, not descended. Only the files directly in the directory are listed (no recursion); files whose extension no codec claims are skipped, and a directory with no dataset files returns an empty inventory rather than aborting. A dataset file that fails to read aborts with its codec's error, naming the file.

Note: counting `records` reads the file through its codec (the one lossless reader), so members() is an honest count, not a header guess; for a large directory it reads every dataset.

Value

A `<artoo_members>` *data frame*, one row per dataset, with columns `file` (source basename), `member` (dataset name), `label`, `records` (row count), `variables` (column count), and `format` (the codec format). Empty when a directory holds no dataset files, and likewise when `format` excludes every one it holds. It is an ordinary data frame underneath.

See Also

Members of one XPORT file: `xpt_members()`.

Per-variable attributes: `columns()` for one dataset's variable pane.

Examples

```
dm <- apply_spec(cdisc_dm, sdtm_spec, "DM", conformance = "off")

# ---- Example 1: one dataset in a file ----
#
# A single-dataset format reports exactly one member.
p <- tempfile(fileext = ".json")
write_json(dm, p)
members(p)

# ---- Example 2: every dataset in a directory ----
#
# Point members() at a folder to inventory each dataset file it holds, one
# row per dataset, dispatched by extension.
dir <- tempfile("datasets")
```

```
dir.create(dir)
write_json(dm, file.path(dir, "dm.json"))
write_rds(dm, file.path(dir, "dm.rds"))
members(dir)

# ---- Example 3: one dataset, two formats, one of them wanted ----
#
# The same dataset stored twice is two rows, because the inventory reports
# what is on disk. Name the format to see only that half.
members(dir, format = "json")
```

`read_dataset` *Read a dataset from any supported format*

Description

Read a clinical file back to a data frame, restoring its `artoo_meta`. The codec is chosen from the file extension (or an explicit `format`), and the metadata the file carries is re-attached, so a value written by `write_dataset()` round-trips losslessly. This is the ingest end of the I/O layer; the per-format wrappers like `read_rds()` call it.

Usage

```
read_dataset(path, format = NULL, col_select = NULL, n_max = Inf, ...)
```

Arguments

`path`	*Source file path.* `<character(1)>: required`. Its extension selects the codec unless `format` is given.
`format`	*Force a codec instead of inferring from the extension.* `<character(1)> | NULL`. One of the registered formats (see `artoo_formats()`).
`col_select`	*Variables to read.* `<character> | NULL`. `NULL` (default) reads every column; otherwise a vector of variable names. Columns return in file order (not the requested order) and the `artoo_meta` is filtered to match. Works on every format: parquet narrows columns natively, the rest filter after decode. **Note:** an unknown name is a `artoo_error_input`, never a silent drop.
`n_max`	*Maximum records to read.* `<numeric(1)>: default Inf`. Caps the row count; the returned `artoo_meta` reports the rows actually read. xpt v8 bounds the disk read; the other formats cap after decode.
`...`	*Codec-specific arguments* passed through to the decoder (see the per-format wrappers, e.g. `read_xpt()`). An argument the codec does not know is an error, never silently ignored.

Value

A `<data.frame>` carrying `artoo_meta` when the file recorded it (read it with `get_meta()`). A file whose payload is not a data frame is a `artoo_error_codec`.

See Also

`write_dataset()` for the inverse; `read_rds()` for the per-format wrapper.

Examples

```
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)

# ---- Example 1: round-trip a dataset through rds ----
#
# Write a conformed dataset, then read it back; the metadata survives.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".rds")
write_dataset(adsl, path)
back <- read_dataset(path)
identical(get_meta(back)@columns, get_meta(adsl)@columns)

# ---- Example 2: the metadata names the dataset and row count ----
#
# The restored artoo_meta exposes the dataset-level attributes.
get_meta(back)@dataset$records
```

`read_json` *Read a dataset from CDISC Dataset-JSON*

Description

Read a CDISC Dataset-JSON v1.1 (`.json`) file back to a data frame, restoring the full `artoo_meta` it carries and realizing SAS date/datetime/time variables to R `Date` / `POSIXct` / `hms::hms`. Column types are reconstructed from the recorded metadata, not guessed from the JSON tokens, so the round-trip is lossless. The ingest end of the I/O layer; a thin wrapper over `read_dataset()` with `format = "json"`.

Usage

```
read_json(path, col_select = NULL, n_max = Inf, encoding = NULL)
```

Arguments

`path`	*Source* `.json` *path.* `<character(1)>`: `required`. A JSON file that is not Dataset-JSON v1.1 aborts with `artoo_error_codec`.
`col_select`	*Variables to read.* `<character> | NULL`. `NULL` (default) reads every column; otherwise a vector of variable names. Columns return in file order (not the requested order) and the `artoo_meta` is filtered to match. Works on every format: parquet narrows columns natively, the rest filter after decode. **Note:** an unknown name is a `artoo_error_input`, never a silent drop.

n_max — *Maximum records to read.* <numeric(1)>: default Inf. Caps the row count; the returned artoo_meta reports the rows actually read. xpt v8 bounds the disk read; the other formats cap after decode.

encoding — *Source charset of the file bytes.* <character(1)> | NULL. NULL (default) reads UTF-8, as Dataset-JSON requires. Pass an IANA or SAS charset name (e.g. "windows-1252") only to read a non-conformant file a producer wrote in that charset; the bytes are transcoded to UTF-8 on read.

Tip: any SAS or IANA spelling listed by artoo_encodings() is accepted.

Value

A <data.frame> carrying artoo_meta (read it with get_meta()).

See Also

write_json() for the inverse; read_dataset() for the generic dispatcher.

Examples

```
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)

# ---- Example 1: round-trip a conformed dataset through Dataset-JSON ----
#
# The variable labels, types, and keys survive the round-trip.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".json")
write_json(adsl, path)
back <- read_json(path)
identical(get_meta(back)@columns, get_meta(adsl)@columns)

# ---- Example 2: the metadata names the dataset and row count ----
#
# The restored artoo_meta exposes the dataset-level attributes.
get_meta(back)@dataset$records
```

---

read_ndjson — *Read a dataset from CDISC Dataset-JSON NDJSON*

Description

Read a newline-delimited CDISC Dataset-JSON v1.1 (.ndjson) file back to a data frame, restoring the full artoo_meta from its metadata line and realizing SAS date/datetime/time variables to R Date / POSIXct / hms::hms. Rows are parsed in bounded slabs, and n_max stops the line loop early, so a partial read of a huge file never parses the tail. A thin wrapper over read_dataset() with format = "ndjson".

Usage

```
read_ndjson(path, col_select = NULL, n_max = Inf, encoding = NULL)
```

Arguments

path	*Source* `.ndjson` *path.* `<character(1)>: required`. A gzip stream (`.ndjson.gz`) is inflated transparently. A file whose first line is not the Dataset-JSON metadata object aborts with `artoo_error_codec`.
col_select	*Variables to read.* `<character> | NULL`. `NULL` (default) reads every column; otherwise a vector of variable names. Columns return in file order (not the requested order) and the `artoo_meta` is filtered to match. Works on every format: parquet narrows columns natively, the rest filter after decode. **Note:** an unknown name is a `artoo_error_input`, never a silent drop.
n_max	*Maximum records to read.* `<numeric(1)>: default Inf`. Caps the row count; the returned `artoo_meta` reports the rows actually read. xpt v8 bounds the disk read; the other formats cap after decode.
encoding	*Source charset of the file bytes.* `<character(1)> | NULL`. `NULL` (default) reads UTF-8, as Dataset-JSON requires. Pass an IANA or SAS charset name (e.g. `"windows-1252"`) only to read a non-conformant file a producer wrote in that charset; each line is transcoded to UTF-8 on read, preserving the bounded `n_max` streaming.

Value

A `<data.frame>` carrying `artoo_meta` (read it with `get_meta()`).

See Also

`write_ndjson()` for the inverse; `read_json()` for the array-form file; `read_dataset()` for the generic dispatcher.

Examples

```
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)

# ---- Example 1: round-trip a conformed dataset through NDJSON ----
#
# The variable labels, types, and keys survive the round-trip.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".ndjson")
write_ndjson(adsl, path)
back <- read_ndjson(path)
identical(get_meta(back)@columns, get_meta(adsl)@columns)

# ---- Example 2: a bounded partial read of the first rows ----
#
# n_max stops the line loop as soon as enough rows are in.
head_rows <- read_ndjson(path, n_max = 5)
get_meta(head_rows)@dataset$records
```

read_parquet *Read a dataset from Apache Parquet*

Description

Read an Apache Parquet (`.parquet`) file back to a data frame, restoring the `artoo_meta` from its `metadata_json` sidecar and realizing SAS date/datetime/time variables to R `Date` / `POSIXct` / `hms::hms`. A parquet written by another tool (with no artoo sidecar) reads back as a bare frame. A thin wrapper over `read_dataset()` with `format = "parquet"`. Requires the lightweight `nanoparquet` package.

Usage

```
read_parquet(path, col_select = NULL, n_max = Inf, encoding = NULL)
```

Arguments

`path`	*Source* `.parquet` *path.* `<character(1)>: required`.
`col_select`	*Variables to read.* `<character> | NULL`. `NULL` (default) reads every column; otherwise a vector of variable names. Columns return in file order (not the requested order) and the `artoo_meta` is filtered to match. Works on every format: parquet narrows columns natively, the rest filter after decode. **Note:** an unknown name is a `artoo_error_input`, never a silent drop.
`n_max`	*Maximum records to read.* `<numeric(1)>: default Inf`. Caps the row count; the returned `artoo_meta` reports the rows actually read. xpt v8 bounds the disk read; the other formats cap after decode.
`encoding`	*Source charset of the string columns.* `<character(1)> | NULL`. `NULL` (default) reads the UTF-8 bytes parquet stores. Pass a charset name only to read a foreign file whose string columns hold that charset's bytes; they are transcoded to UTF-8 on read. **Tip:** any SAS or IANA spelling listed by `artoo_encodings()` is accepted.

Value

A `<data.frame>` carrying `artoo_meta` when the file recorded it (read it with `get_meta()`); otherwise a plain data frame.

See Also

`write_parquet()` for the inverse; `read_dataset()` for the generic dispatcher.

Examples

```
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)

# ---- Example 1: round-trip a conformed dataset through Parquet ----
#
```

```
# The variable labels, types, and keys survive the round-trip.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".parquet")
write_parquet(adsl, path)
back <- read_parquet(path)
get_meta(back)@columns$STUDYID$label

# ---- Example 2: the metadata names the dataset and row count ----
#
# The restored artoo_meta exposes the dataset-level attributes.
get_meta(back)@dataset$records
```

`read_rds` *Read a dataset from rds*

Description

Read an R `.rds` file written by `write_rds()` (or any rds carrying a `metadata_json` attribute) back to a data frame with its `artoo_meta` restored. A thin wrapper over `read_dataset()` with `format = "rds"`.

Usage

```
read_rds(path, col_select = NULL, n_max = Inf, encoding = NULL)
```

Arguments

`path`	*Source* `.rds` *path.* `<character(1)>: required`.
`col_select`	*Variables to read.* `<character> | NULL`. `NULL` (default) reads every column; otherwise a vector of variable names. Columns return in file order (not the requested order) and the `artoo_meta` is filtered to match. Works on every format: parquet narrows columns natively, the rest filter after decode. **Note:** an unknown name is a `artoo_error_input`, never a silent drop.
`n_max`	*Maximum records to read.* `<numeric(1)>: default Inf`. Caps the row count; the returned `artoo_meta` reports the rows actually read. xpt v8 bounds the disk read; the other formats cap after decode.
`encoding`	*Source charset of the string columns.* `<character(1)> | NULL`. `NULL` (default) returns the strings exactly as saved (faithful R round-trip). Pass a charset name only to transcode a foreign rds whose string columns hold that charset's bytes. **Tip:** any SAS or IANA spelling listed by `artoo_encodings()` is accepted.

Value

A `<data.frame>` carrying `artoo_meta` when the file recorded it. An rds holding anything other than a data frame is a `artoo_error_codec`; use `readRDS()` for arbitrary objects.

See Also

`write_rds()` for the inverse; `read_dataset()` for the generic dispatcher.

Examples

```
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists
)

# ---- Example 1: read a dataset written by write_rds() ----
#
# The restored frame carries the same metadata it was written with.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".rds")
write_rds(adsl, path)
back <- read_rds(path)
get_meta(back)@dataset$records

# ---- Example 2: a plain rds still reads as a data frame ----
#
# An rds without artoo metadata reads back as an ordinary frame.
bare <- tempfile(fileext = ".rds")
saveRDS(cdisc_dm, bare)
nrow(read_rds(bare))
```

`read_spec` *Read a specification from JSON, Excel, or Define-XML*

Description

Read a clinical-dataset specification into a validated `artoo_spec`, dispatching on the file extension: artoo's native JSON (the inverse of `write_spec()`), a Pinnacle 21 (P21) Excel workbook, or a native Define-XML 2.0/2.1 document. The returned spec is the lingua franca the rest of artoo applies and serialises.

Usage

```
read_spec(path, datasets = NULL, on_duplicate = c("error", "first", "warn"))
```

Arguments

`path`	*The specification file to read.* `<character(1)>: required`. A `.json` (native), `.xlsx` / `.xls` (P21), or `.xml` (Define-XML 2.0 or 2.1) file. **Requirement:** reading a P21 workbook needs the `readxl` package, and reading a define.xml needs `xml2`.

`datasets` *Read only these datasets.* `<character> | NULL`. `NULL` (default) reads the whole spec. Otherwise the spec is scoped to the named datasets before validation, so one broken sheet elsewhere in a workbook cannot block the dataset you are working on. An unknown name aborts listing what the file defines.

`on_duplicate` *Policy for a variable defined more than once.* `<character(1)>`. A workbook row duplicated within one dataset makes the spec ambiguous; the finding is reported with its source location (sheet and row numbers for Excel). One of:

- `"error"` (default) abort, naming each duplicate's rows.
- `"first"` keep the first definition of each, dropping the rest with a message.
- `"warn"` keep the first definition and warn (`artoo_warning_spec`).

Details

Three formats, one validator. A `.json` file is read as artoo native JSON; a `.xlsx` / `.xls` file is read as a P21 workbook; a `.xml` file is read as Define-XML 2.x. Either way the result is built through `artoo_spec()`, so type canonicalisation and cross-slot integrity checks are identical regardless of source.

Define-XML ingestion (needs the `xml2` package). ItemGroupDefs become datasets (keys derived from the ItemRef KeySequence), ItemRef + ItemDef pairs become variables, CodeLists become codelists (`def:ExtendedValue = "Yes"` marks an extended term), MethodDefs / CommentDefs / leaves become the supporting slots, and ValueListDefs land in the value-level slot with their where-clauses rendered as readable text.

Note: an `ExternalCodeList` (MedDRA, ISO-3166) names a dictionary rather than an enumerable membership list, so it lands in `dictionaries` rather than `codelists`; a variable that references one keeps the reference, because a workbook has one column for both. Define-XML v1.0 (the 2005 model) is refused with guidance.

P21 ingestion. Sheets are located by a tolerant alias match (case-, space-, and spelling-variant insensitive). Datasets and Variables are required; Codelists and ValueLevel are optional (the latter becomes the spec's value-level slot). Every cell is read as text, then the dataset and codelist foreign keys are forward-filled to recover merged cells (which the Excel reader returns as `NA` on continuation rows). A key that cannot be resolved aborts with `artoo_error_spec` rather than being silently dropped.

Value

A validated `artoo_spec`. Inspect it with `spec_datasets()` / `spec_variables()`, check it with `validate_spec()`, or persist it with `write_spec()`.

See Also

Inverse: `write_spec()` serialises a spec to native JSON.

Build / inspect: `artoo_spec()`, `spec_datasets()`, `spec_variables()`, `validate_spec()`.

Examples

```
# ---- Example 1: round-trip a spec through native JSON ----
#
```

```
# write_spec() and read_spec() are inverses on the JSON path: the spec
# that comes back is identical to the one written.
spec <- artoo_spec(cdisc_sdtm_datasets, cdisc_sdtm_variables, codelists = cdisc_codelists)
path <- tempfile(fileext = ".json")
write_spec(spec, path)
back <- read_spec(path)
identical(back, spec)

# ---- Example 2: scope the read to one dataset ----
#
# `datasets =` reads just the domain you are working on — validation is
# scoped with it, so a problem elsewhere in the workbook cannot block
# this dataset.
dm_spec <- read_spec(path, datasets = "DM")
spec_datasets(dm_spec)
head(spec_variables(dm_spec, "DM")[, c("variable", "label", "data_type")])
```

`read_xpt` *Read a dataset from SAS XPORT*

Description

Read a SAS Transport (`.xpt`) file (v5 or v8) back to a data frame, restoring the `artoo_meta` its NAMESTR records carry and realizing SAS date/datetime/time variables to R `Date` / `POSIXct` / `hms::hms`. The ingest end of the I/O layer; a thin wrapper over `read_dataset()` with `format = "xpt"`.

Usage

```
read_xpt(path, encoding = NULL, col_select = NULL, n_max = Inf, member = NULL)
```

Arguments

`path`	*Source* `.xpt` *path.* `<character(1)>: required`.
`encoding`	*Force a source charset.* `<character(1)> | NULL`. `NULL` (default) auto-detects (UTF-8 when every character value and label is valid UTF-8, else Windows-1252). IANA and SAS names both work. **Tip:** any SAS or IANA spelling listed by `artoo_encodings()` is accepted.
`col_select`	*Variables to read.* `<character> | NULL`. `NULL` (default) reads every column; otherwise a vector of variable names (matching the names as stored, uppercase for v5). Columns return in file order, and the `artoo_meta` is filtered to match. **Note:** an unknown name is a `artoo_error_input`, never a silent drop.
`n_max`	*Maximum records to read.* `<numeric(1)>: default Inf`. Caps the row count; the returned `artoo_meta` reports the rows actually read.

member — *Which member of a multi-member transport file to read.* <character(1) | numeric(1)> | NULL. A transport file can hold several datasets; pass a member name (case-insensitive) or 1-based index to pick one. NULL (default) reads a single-member file directly and aborts on a multi-member file, pointing at xpt_members().
Tip: xpt_members(path) lists what a file holds before you choose.

Details

The character encoding is auto-detected (UTF-8 if every character value is valid UTF-8, else Windows-1252) and recorded on the returned artoo_meta, so a later write_xpt() reproduces it; pass encoding to override. XPORT cannot record its own encoding, so this detection is a heuristic. See write_xpt() for what XPORT can and cannot preserve.

Value

A <data.frame> carrying artoo_meta (read it with get_meta()).

See Also

xpt_members() to list a file's members; write_xpt() for the inverse; read_dataset() for the generic dispatcher.

Examples

```
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists
)

# ---- Example 1: round-trip a conformed dataset through xpt ----
#
# Write ADSL, read it back; the variable labels and lengths survive.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".xpt")
write_xpt(adsl, path)
back <- read_xpt(path)
get_meta(back)@columns$STUDYID$label

# ---- Example 2: pick one member of a multi-member transport file ----
#
# Build a two-member file by concatenating two single-member files (every
# member section is 80-byte padded), then read one dataset out of it.
dm <- apply_spec(cdisc_dm, sdtm_spec, "DM", conformance = "off")
p_dm <- tempfile(fileext = ".xpt")
write_xpt(dm, p_dm)
multi <- tempfile(fileext = ".xpt")
writeBin(
  c(
    readBin(path, "raw", file.size(path)),
    readBin(p_dm, "raw", file.size(p_dm))[-(1:240)]
  ),
  multi
```

```
)
xpt_members(multi)$name
nrow(read_xpt(multi, member = "DM"))
```

`repair_spec` *Repair a spec from its conformance findings*

Description

Take the `integer_fraction` and `integer_overflow` findings a `check_spec()` or `check_study()` run reports and return a new spec with every offending variable retyped to `"float"`, so a frame that the original spec would refuse to coerce now conforms. This closes the loop on the spec-side fix: inspect the findings, then apply them all at once instead of editing the source workbook variable by variable. Persist the result with `write_spec()`.

Usage

```
repair_spec(spec, findings)
```

Arguments

`spec`	*The specification to repair.* `<artoo_spec>: required`.
`findings`	*A findings data frame.* `<data.frame>: required`. The result of `check_spec()` or `check_study()`; must carry the `check`, `dataset`, and `variable` columns.

Details

Scope. Only the two lossy-integer findings are repaired (`integer_fraction`, `integer_overflow`) — both mean "the spec says `integer` but the data is not", and `"float"` is the loss-free fix. Other findings are ignored; this is not a general spec rewriter. When no repairable finding is present the spec is returned unchanged, with a note.

Built on `set_type()`. Each `(dataset, variable)` pair is applied through the same validated override `set_type()` uses, so the result is a fully re-validated `artoo_spec`, never a hand-edited internal.

Value

A new `<artoo_spec>` with the flagged variables retyped to `"float"`, or `spec` unchanged when there is nothing to repair. The input is never mutated.

See Also

Primitive: `set_type()` to retype a chosen variable directly.

Findings: `check_spec()` for one dataset, `check_study()` across a study. **Persist:** `write_spec()`.

Examples

```
# ---- Example 1: auto-repair an integer/fractional mismatch ----
#
# adam_spec types ADSL.AGE as integer. Give it fractional ages and
# check_spec() raises an integer_fraction error; repair_spec() flips AGE
# (and only AGE) to float, and the corrected spec then applies cleanly.
dat <- cdisc_adsl
dat$AGE <- dat$AGE + 0.5
findings <- check_spec(dat, adam_spec, "ADSL")
fixed <- repair_spec(adam_spec, findings)
spec_variables(fixed, "ADSL")$data_type[
  spec_variables(fixed, "ADSL")$variable == "AGE"
]

# ---- Example 2: nothing to repair is a no-op ----
#
# The bundled data conforms, so its findings carry no integer_fraction or
# integer_overflow rows and the spec is returned unchanged.
clean <- check_spec(cdisc_adsl, adam_spec, "ADSL")
identical(repair_spec(adam_spec, clean), adam_spec)
```

---

set_meta *Attach metadata to a dataset*

Description

Stamp a `artoo_meta` onto a data frame as a single Dataset-JSON string in its `metadata_json` attribute. Every `write_*()` codec reads that string back with `get_meta()` and embeds it verbatim, so the metadata survives the trip to any format. Use it to attach metadata to a bare frame before a write, or to re-stamp after a tidyverse verb has dropped attributes.

Usage

```
set_meta(x, meta)
```

Arguments

`x`	*The data frame to stamp.* `<data.frame>: required`.
`meta`	*The metadata to attach.* `<artoo_meta>: required`. Usually from `get_meta()` or built by `apply_spec()`.

Value

The data frame `x`, with its `metadata_json` attribute set. Pass it on to a `write_*()` codec or back through `get_meta()`.

See Also

get_meta() for the read half; apply_spec() which stamps it.

Examples

```
# ---- Example 1: re-stamp metadata a dplyr verb would drop ----
#
# Conform a dataset, capture its metadata, then re-attach after an
# attribute-dropping transform so the write stays lossless.
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists
)
adsl <- apply_spec(cdisc_adsl, spec, "ADSL")
meta <- get_meta(adsl)
trimmed <- head(as.data.frame(adsl), 5)
attr(trimmed, "metadata_json") <- NULL
set_meta(trimmed, meta)

# ---- Example 2: borrow metadata from a conformed dataset ----
#
# A writer with a raw frame can lift metadata off a conformed dataset and
# stamp it onto the bare frame (DM is SDTM, so its spec is sdtm-shaped).
sdtm <- artoo_spec(
  cdisc_sdtm_datasets, cdisc_sdtm_variables,
  codelists = cdisc_codelists
)
meta_dm <- get_meta(apply_spec(cdisc_dm, sdtm, "DM"))
dm <- set_meta(cdisc_dm, meta_dm)
is_artoo_meta(get_meta(dm))
```

---

set_type *Override a variable's dataType in a spec*

Description

Return a new artoo_spec with one or more variables retyped. This is the supported, in-R way to correct a spec when the data disagrees with its declared dataType (e.g. a variable typed integer whose extract holds fractional values): fix it here rather than editing the source workbook, then drive apply_spec() with the corrected spec. The spec is immutable, so the original is never changed.

Usage

```
set_type(spec, dataset, ...)
```

Arguments

spec	*The specification to amend.* <artoo_spec>: required.
dataset	*The dataset whose variables to retype.* <character(1)>: required. Must name a dataset in spec.
...	*Named* variable = type *pairs.* Each name is a variable in dataset; each value is a CDISC dataType ("string", "integer", "decimal", "float", "double", "boolean", "date", "datetime", "time", "URI") or a recognised spelling of one. At least one pair is required and every argument must be named. **Tip:** to undo an integer dataType that the data does not satisfy, set "float" (IEEE double) or "decimal" (exact, exchanged as text).

Details

Per-dataset scope. A type is set only on the named dataset's row. A variable that appears in several datasets keeps its other rows' types; call set_type() once per dataset to change them all. Spec-wide consequences (a variable typed inconsistently across datasets) are a validate_spec() concern, not a construction error here.

Canonicalised, then validated. Each supplied type is mapped through the closed CDISC dataType vocabulary, so "Float", "decimal", and "text" all resolve; an unrecognised token aborts with artoo_error_type. The rebuilt spec is re-validated, so an override that would break the spec aborts with artoo_error_spec.

Value

A new <artoo_spec> with the named variables retyped, ready for apply_spec() or write_spec(). The input spec is unchanged.

See Also

Auto-repair: repair_spec() to apply every integer_fraction / integer_overflow fix from a findings frame at once.

Workflow: apply_spec() to conform with the corrected spec; write_spec() to persist it; check_spec() to find the mismatches.

Examples

```
# ---- Example 1: retype one variable the data disagrees with ----
#
# The bundled adam_spec types ADSL.AGE as integer. If an extract stored it
# with fractional values, retype it to float so apply_spec() coerces
# without loss. set_type() returns a new spec; the original is untouched.
fixed <- set_type(adam_spec, "ADSL", AGE = "float")
v <- spec_variables(fixed, "ADSL")
v[v$variable == "AGE", c("variable", "data_type")]

# ---- Example 2: retype several at once, original left intact ----
#
# Pass any number of variable = type pairs; canonical dataTypes and common
```

```
# spellings both resolve. The source spec is immutable, so adam_spec still
# reports AGE as its original type.
patched <- set_type(adam_spec, "ADSL", AGE = "decimal", TRTSDT = "date")
spec_variables(adam_spec, "ADSL")$data_type[
  spec_variables(adam_spec, "ADSL")$variable == "AGE"
]
```

`spec_codelists` *Codelist terms*

Description

Return the controlled-terminology terms and decodes a spec carries: one codelist's terms when `codelist_id` names it, or the full `codelists` slot when `codelist_id` is `NULL`. Use it to inspect the values a coded variable is allowed to take before applying the spec. Mirrors the `spec_variables()` filter pattern.

Usage

```
spec_codelists(spec, codelist_id = NULL)
```

Arguments

`spec`	*The specification to read.* `<artoo_spec>: required`.
`codelist_id`	*The codelist to return.* `<character(1)> | NULL`. When `NULL` (default) the whole codelists table is returned. **Restriction:** a non-`NULL` id must name a codelist present in the spec's `codelists` slot; an unknown id aborts with `artoo_error_input`.

Value

A data frame of codelist terms, one row per term: every term when `codelist_id` is `NULL`, else the named codelist's terms. Columns:

- `codelist_id` — the codelist identifier variables reference.
- `term` — the submission value (what conformed data carries).
- `decode` — the human-readable decoded value.
- `order` — display order within the codelist.
- `extended` — `TRUE` marks an extensible codelist (sponsor terms allowed; non-members downgrade to notes in `check_spec()`).
- `comment_id` — reference into the comments slot.

See Also

`spec_variables()` for which variables reference a codelist.

Examples

```
# ---- Example 1: the terms behind a coded variable ----
#
# SEX is coded against C66731; spec_codelists() returns the terms and their
# decodes that apply_spec() will enforce or decode.
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists
)
spec_codelists(spec, "C66731")

# ---- Example 2: the whole codelists table ----
#
# Called with no id, it returns every term across every codelist.
head(spec_codelists(spec))
```

`spec_comments` *Comment definitions in a spec*

Description

Return the comment definitions a specification carries. Datasets, variables, value-level rows, and codelists reference these by `comment_id`; `validate_spec()` checks the references resolve and each referenced comment has a body.

Usage

```
spec_comments(spec)
```

Arguments

`spec`	*The specification to read.* `<artoo_spec>: required`.

Value

A data frame of comment metadata, one row per comment, with all four columns: `comment_id`, `description`, `document_id`, `pages`. Empty when the spec defines no comments.

See Also

`spec_methods()`, `spec_documents()`, `validate_spec()`.

Examples

```
# ---- Example 1: the comments a spec defines ----
#
# Build a spec with one comment and read it back.
spec <- artoo_spec(
  data.frame(dataset = "ADSL"),
  data.frame(dataset = "ADSL", variable = "AGE", data_type = "integer"),
  comments = data.frame(
    comment_id = "C.AGE",
    description = "Age in years at informed consent.",
    stringsAsFactors = FALSE
  )
)
spec_comments(spec)
```

`spec_datasets` *Dataset names in a spec*

Description

List the datasets a specification defines. The result is the set of names you pass as the `dataset` argument to the other accessors and to `apply_spec()`.

Usage

```
spec_datasets(spec)
```

Arguments

`spec`	*The specification to read.* `<artoo_spec>`: `required`.

Value

A character vector of dataset names, de-duplicated and with `NA`s dropped. Empty when the spec has no datasets.

See Also

`spec_variables()` for one dataset's variables; `spec_keys()` for its sort keys.

Examples

```
# ---- Example 1: the datasets the pilot ADaM spec defines ----
#
# Build the spec from the bundled CDISC-pilot tables and list its
# datasets — the names you pass to the other accessors.
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
```

```
  codelists = cdisc_codelists
)
spec_datasets(spec)
```

`spec_documents` *Document references in a spec*

Description

Return the document references a specification carries. Methods and comments point to these by `document_id`.

Usage

```
spec_documents(spec)
```

Arguments

`spec`	*The specification to read.* `<artoo_spec>: required`.

Value

A data frame of document metadata (`document_id`, `title`, `href`), one row per document. Empty when the spec defines none.

See Also

`spec_methods()`, `spec_comments()`, `validate_spec()`.

Examples

```
# ---- Example 1: the documents a spec defines ----
#
# Build a spec with one document reference and read it back.
spec <- artoo_spec(
  data.frame(dataset = "ADSL"),
  data.frame(dataset = "ADSL", variable = "AGE", data_type = "integer"),
  documents = data.frame(
    document_id = "SAP",
    title = "Statistical Analysis Plan",
    stringsAsFactors = FALSE
  )
)
spec_documents(spec)
```

spec_keys *Sort keys for a dataset*

Description

Parse a dataset's sort keys into a character vector of variable names. These keys drive the sort step of `apply_spec()` and the `keySequence` written to each output format.

Usage

```
spec_keys(spec, dataset)
```

Arguments

`spec`	*The specification to read.* `<artoo_spec>: required`.
`dataset`	*The dataset whose keys to parse.* `<character(1)>: required`. **Restriction:** must name a dataset in the spec.

Value

A character vector of key variable names, split from the dataset's `keys` cell (whitespace- or comma-separated). Empty when no keys are declared.

See Also

`spec_datasets()` for the dataset names; `spec_variables()` for the variables a key must reference.

Examples

```
# ---- Example 1: parse a dataset's sort keys ----
#
# Declare DM's keys, then read them back as the ordered vector apply_spec()
# sorts by. (STUDYID and USUBJID are real DM variables in the demo data.)
ds <- cdisc_sdtm_datasets
ds$keys[ds$dataset == "DM"] <- "STUDYID USUBJID"
spec <- artoo_spec(ds, cdisc_sdtm_variables, codelists = cdisc_codelists)
spec_keys(spec, "DM")
```

spec_methods *Derivation methods in a spec*

Description

Return the method definitions a specification carries. Variables and value-level rows reference these by `method_id`; `validate_spec()` checks that every reference resolves and that each referenced method is complete (has a description).

Usage

```
spec_methods(spec)
```

Arguments

spec	*The specification to read.* `<artoo_spec>: required`.

Value

A data frame of method metadata, one row per method, with all eight columns: `method_id`, `description`, `name`, `type`, `expression_context`, `expression_code`, `document_id`, `pages`. Empty when the spec defines no methods.

See Also

`spec_comments()`, `spec_documents()`, `validate_spec()`.

Examples

```
# ---- Example 1: the methods a spec defines ----
#
# Build a spec with one derivation method and read it back.
spec <- artoo_spec(
  data.frame(dataset = "ADSL"),
  data.frame(dataset = "ADSL", variable = "AGEGR1", data_type = "string"),
  methods = data.frame(
    method_id = "MT.AGEGR1",
    description = "Age group from AGE.",
    stringsAsFactors = FALSE
  )
)
spec_methods(spec)
```

spec_standard *The CDISC standard a spec implements*

Description

Return the one CDISC standard the specification carries (e.g. `"ADaMIG 1.1"`, `"SDTMIG 3.2"`). A `artoo_spec` is single-standard by construction — `artoo_spec()` aborts when its sources mix standards — so this is always a scalar; `NA` when no source named one.

Usage

```
spec_standard(spec)
```

Arguments

`spec`	*The specification to read.* `<artoo_spec>: required`.

Value

A `<character(1)>`: the standard, or `NA` when unspecified.

See Also

`spec_study()` for the rest of the study-level metadata; `artoo_spec()` for how the standard is resolved.

Examples

```
# ---- Example 1: the standard set at construction ----
#
# Pass the standard explicitly (or let it resolve from a P21 workbook's
# Standard column / a Define-XML study block) and read it back.
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists,
  standard = "ADaMIG 1.1"
)
spec_standard(spec)

# ---- Example 2: unspecified resolves to NA ----
#
# A spec built without any standard source carries NA.
bare <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists
)
spec_standard(bare)
```

spec_study *Study-level metadata*

Description

Return the study-level metadata row, or a single field from it. Holds the canonical study fields (study_name, study_description, protocol_name — every source spelling is canonicalised by artoo_spec()) plus any other study-scoped fields a source provides (the CDISC standard lives on its own property — see spec_standard()).

Usage

```
spec_study(spec, field = NULL)
```

Arguments

spec	*The specification to read.* <artoo_spec>: required.
field	*Return one field instead of the row.* <character(1)> | NULL. When NULL (default) the whole study data frame is returned. **Restriction:** a non-NULL field must be a column of the study table; an unknown field aborts with artoo_error_input.

Value

The study data frame (one row), or the value of one field. The canonical fields are study_name, study_description, and protocol_name — every source spelling is canonicalised to these by artoo_spec() — plus any other field the source carried verbatim (e.g. define_version from a Define-XML read).

See Also

spec_datasets() for the datasets the study scopes; spec_standard() for the spec's CDISC standard.

Examples

```
# ---- Example 1: the whole study row, then one field ----
#
# spec_study() with no field returns the study-level table; pass a field
# name to pull a single value such as the study name.
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists,
  study = data.frame(study_name = "CDISCPILOT01")
)
spec_study(spec)
spec_study(spec, "study_name")
```

`spec_variables` *Variables in a spec*

Description

Return the variable-metadata table for one dataset, or for the whole spec. Each row carries the variable's CDISC `data_type`, label, length, display format, key sequence, and codelist reference.

Usage

```
spec_variables(spec, dataset = NULL)
```

Arguments

`spec`	*The specification to read.* `<artoo_spec>: required`.
`dataset`	*Restrict to one dataset.* `<character(1)> | NULL`. When `NULL` (default) every dataset's variables are returned; otherwise only the named dataset's rows. **Restriction:** a non-`NULL` `dataset` must name a dataset in the spec (see `spec_datasets()`); an unknown name aborts with `artoo_error_input`.

Value

A data frame of variable metadata, one row per variable, with 22 columns (absent ones are filled with typed `NA` at construction):

- `dataset`, `variable` — the identifying pair (unique within a spec).
- `itemoid` — the Define-XML / Dataset-JSON item OID, when recorded.
- `label` — the variable label (<= 40 bytes for XPORT v5).
- `data_type` — canonical CDISC dataType (`string`, `integer`, `decimal`, `float`, `double`, `boolean`, `date`, `datetime`, `time`, `URI`).
- `target_data_type` — `integer`/`decimal` when a temporal variable stores as a SAS-epoch numeric; `NA` means ISO 8601 text (`--DTC`).
- `length` — declared storage length (bytes for character).
- `display_format`, `informat` — SAS format / informat strings.
- `key_sequence` — 1-based position in the dataset sort key.
- `order` — column position in the dataset.
- `codelist_id`, `method_id`, `comment_id` — references into the codelists / methods / comments slots.
- `mandatory` — logical obligation flag (`NA` is treated as mandatory by `check_spec()`).
- `significant_digits` — for `decimal` variables.
- `origin`, `source`, `predecessor`, `assigned_value`, `pages`, `role` — Define-XML provenance fields, carried as-is.

Filter or arrange it with ordinary base / `dplyr` verbs.

See Also

spec_datasets() for the dataset names; spec_codelists() for a variable's controlled terminology.

Examples

```
spec <- artoo_spec(cdisc_sdtm_datasets, cdisc_sdtm_variables, codelists = cdisc_codelists)

# ---- Example 1: one dataset's variables ----
#
# Pass a dataset name to get just that domain's variables, already
# canonicalised to CDISC dataTypes.
head(spec_variables(spec, "DM")[, c("variable", "label", "data_type")])

# ---- Example 2: every variable across the spec ----
#
# Omit `dataset` to get the full table, e.g. to count variables per domain.
table(spec_variables(spec)$dataset)
```

---

sync_meta *Re-align metadata with a transformed data frame*

Description

Re-attach and reconcile a artoo_meta after a transformation that dropped or reshaped it: the metadata's columns are narrowed and reordered to the frame's current columns, the record count is refreshed, the keys are recomputed, and a column the metadata does not describe gets an entry synthesized from its class and attributes. The one-liner to run after a dplyr (or base) pipeline, before handing the frame to a write_*() codec.

Usage

```
sync_meta(x, meta = NULL)
```

Arguments

x	*The transformed data frame.* <data.frame>: required.
meta	*The metadata to reconcile against.* <artoo_meta> | NULL. NULL (default) uses the frame's own metadata_json attribute. **Requirement:** when the transform dropped the attribute (base [subsetting does), capture get_meta() before the pipeline and pass it here; a bare frame with no meta aborts with artoo_error_input.

Details

Why it exists. Base row subsetting (`x[i, ]`) drops the frame's `metadata_json` attribute, and many tidyverse verbs rebuild the frame. `sync_meta()` takes the last-known metadata (the frame's own attribute when it survived, or an explicit `meta`) and makes it agree with the data again, so the round trip stays lossless without hand-editing.

Value

A `<data.frame>`: `x` re-stamped with the reconciled `artoo_meta`. Hand it to any `write_*()` codec.

See Also

Read / attach: `get_meta()`, `set_meta()`.

Produce conformed frames: `apply_spec()`.

Examples

```
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)

# ---- Example 1: re-attach after an attribute-dropping subset ----
#
# Base subsetting drops the metadata; capture it first, transform, then
# sync. The metadata narrows to the kept columns and the new row count.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
meta <- get_meta(adsl)
elderly <- adsl[adsl$AGE > 65, c("STUDYID", "USUBJID", "AGE")]
synced <- sync_meta(elderly, meta)
get_meta(synced)@dataset$records

# ---- Example 2: a derived column gains a synthesized entry ----
#
# A new column the metadata does not describe is profiled from its class,
# so the frame still writes losslessly.
adsl$AGEGR9 <- ifelse(adsl$AGE > 65, ">65", "<=65")
synced2 <- sync_meta(adsl)
get_meta(synced2)@columns$AGEGR9$dataType
```

`validate_define` *Validate a Define-XML document against its CDISC schema*

Description

Schema-validates a `define.xml` against the Clinical Data Interchange Standards Consortium (CDISC) XML Schema that artoo bundles, offline. Use it on a document from any source, whether or not artoo wrote it, as the first gate before a submission package leaves your hands.

Usage

```
validate_define(path, version = NULL)
```

Arguments

path	*Define-XML document to validate.* <character(1)>: required.
version	*Define-XML version to validate against.* <character(1)> | NULL. NULL (default) reads it from the document's def namespace. Supply "2.0" or "2.1" to assert a version instead, which turns a version mismatch into findings rather than silent success.

Details

The version is detected, not assumed. Define-XML 2.0 and 2.1 declare different def namespaces, so the document says which schema applies. Passing version overrides that, which is how you confirm a file really is the version it claims.

Schema validity is a floor, not a ceiling. A document can satisfy the schema and still be unfit to submit. Two whole classes of defect are invisible here: a reference that points at nothing, and a definition that nothing points at. Neither is expressible in XML Schema. Use lint_define() for those.

Value

An artoo_check *object.* Its @findings data frame has columns check, dimension, severity, dataset, variable, message, and is empty when the document is valid. Print it for the sectioned report.

See Also

Check further: lint_define() for reference integrity, which schema validation cannot see.

Specs: read_spec() reads a Define-XML document into an artoo_spec().

Examples

```
# ---- Example 1: a conforming document ----
#
# artoo bundles both CDISC schema trees, so validation runs offline. The
# bundled minimal example conforms, so the findings table comes back empty
# and the summary records which version was detected.
minimal <- system.file("extdata", "define-minimal.xml", package = "artoo")
report <- validate_define(minimal)
nrow(report@findings)
report@summary[c("define_version", "valid")]

# ---- Example 2: a document that breaks the schema ----
#
# Give ItemGroupDef/@Repeating a value outside its enumeration. The schema
# catches it, and each violation becomes one finding naming the element and
# the line it sits on.
```

```
broken <- tempfile(fileext = ".xml")
writeLines(
  sub('Repeating="No"', 'Repeating="Maybe"', readLines(minimal), fixed = TRUE),
  broken
)
bad <- validate_define(broken)
bad@summary$valid
substr(bad@findings$message[1], 1, 60)
```

`validate_spec` *Validate a specification for submission-readiness*

Description

Run artoo's bundled, self-contained checks over a `artoo_spec`, **scoped to the dataset(s) you are working on**, and return a `artoo_check` that prints a sectioned report. Every finding is keyed to an open rule in the shipped catalog (see `spec_rules.json`); the result object keeps the findings as a plain data frame in `@findings` for programmatic use.

Usage

```
validate_spec(
  spec,
  data = NULL,
  dataset = NULL,
  on_error = c("off", "warn", "abort")
)
```

Arguments

`spec`	*The specification to validate.* `<artoo_spec>: required`.
`data`	*Optional input data for controlled-terminology checks.* `<data.frame> | named list | NULL`. When supplied, data values are cross-checked against the spec codelists. A single data frame requires a length-1 `dataset`; pass a named list to validate several at once.
`dataset`	*Restrict to one or more datasets.* `<character> | NULL`. `NULL` (default) validates every dataset. **Restriction:** each name must be a dataset in the spec.
`on_error`	*What to do with an error-severity finding.* `<character(1)>`. One of: • `"off"` (default) collect and return every finding; never signal. • `"warn"` additionally `cli_warn` (`artoo_warning_validation`) with the error count. • `"abort"` additionally abort with `artoo_error_validation`. All findings are collected and returned in every case.

Details

Dataset-scoped. A spec workbook carries many datasets. Pass `dataset` to validate only the one(s) you are working on — the methods, comments, and codelists those datasets reference are checked for completeness, but unrelated datasets are not. `dataset = NULL` validates the whole spec.

Collect, do not stop. Every finding is collected and returned; `validate_spec()` does not abort on an error-severity finding unless `on_error = "abort"`.

Value

A `artoo_check` *object.* Its `@findings` data frame has columns `check`, `dimension`, `severity`, `dataset`, `variable`, `message`. Print it for the sectioned report.

See Also

`artoo_spec()` to build a spec; `spec_methods()` / `spec_comments()` for the metadata checked.

Examples

```
# ---- Example 1: validate one dataset ----
#
# Build a spec from the bundled ADaM tables and validate it; the
# result prints a sectioned report and keeps the findings table.
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists
)
chk <- validate_spec(spec, dataset = "ADSL")
chk@findings

# ---- Example 2: gate on errors with on_error = "abort" ----
#
# Point a key at a missing variable, then validate with on_error = "abort"
# and catch the resulting error.
bad_ds <- cdisc_sdtm_datasets
bad_ds$keys <- "NOTAVAR"
bad <- artoo_spec(bad_ds, cdisc_sdtm_variables, codelists = cdisc_codelists)
tryCatch(
  validate_spec(bad, dataset = "DM", on_error = "abort"),
  artoo_error_validation = function(e) conditionMessage(e)[1]
)
```

`write_dataset` *Write a dataset to any supported format*

Description

Serialize a data frame to a clinical file format, preserving its `artoo_meta` losslessly. The codec is chosen from the file extension (or an explicit `format`), so one call covers xpt, Dataset-JSON, Parquet, and rds. This is the emit end of the artoo workflow; the per-format wrappers like `write_rds()` are thin sugar over it.

Usage

```
write_dataset(x, path, format = NULL, ...)
```

Arguments

`x`	*The dataset to write.* `<data.frame>: required`. Typically the output of `apply_spec()`, carrying `artoo_meta`.
`path`	*Destination file path.* `<character(1)>: required`. Its extension selects the codec unless `format` is given.
`format`	*Force a codec instead of inferring from the extension.* `<character(1)> | NULL`. One of the registered formats (see `artoo_formats()`).
`...`	*Codec-specific arguments* passed through to the encoder (see the per-format wrappers, e.g. `write_xpt()`, for what each codec accepts). An argument the codec does not know is an error, never silently ignored.

Value

The input `x`, invisibly, so a write can sit mid-pipeline. Called for the side effect of writing `path`.

See Also

`read_dataset()` for the inverse; `write_rds()` for the per-format wrapper; `artoo_formats()` for what is available.

Examples

```
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)

# ---- Example 1: write a conformed dataset, inferring rds from the path ----
#
# apply_spec() attaches the metadata; write_dataset() carries it into the
# file so a later read is lossless.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".rds")
write_dataset(adsl, path)

# ---- Example 2: force the format for an unconventional extension ----
#
# When the extension does not name the format, pass it explicitly.
alt <- tempfile(fileext = ".data")
write_dataset(adsl, alt, format = "rds")
```

write_json *Write a dataset to CDISC Dataset-JSON*

Description

Serialize a data frame to a CDISC Dataset-JSON v1.1 (`.json`) file, Dataset-JSON being the native home of the `artoo_meta` shape: the file is the metadata block plus a flat `rows` array. The emit end of the artoo workflow (spec -> apply_spec -> write_json); a thin wrapper over `write_dataset()` with `format = "json"`.

Usage

```
write_json(
  x,
  path,
  on_invalid = c("error", "translit", "fold", "replace", "ignore"),
  created = NULL,
  strict = FALSE
)
```

Arguments

x	*The dataset to write.* `<data.frame>: required`. Typically the output of `apply_spec()`, carrying `artoo_meta`.
path	*Destination* `.json` *path.* `<character(1)>: required`.
on_invalid	*Policy for values that are not valid UTF-8.* `<character(1)>: default "error"`. One of `"error"` (abort with `artoo_error_codec`, naming the offenders with their invalid bytes hex-escaped), `"replace"` (substitute ? and warn with `artoo_warning_encoding`), `"ignore"` (drop the invalid bytes), or `"translit"` / `"fold"` (like `"error"`; a byte-level invalidity has no character fold, the options exist so one policy value can thread a whole multi-format pipeline). The same policy vocabulary as `write_xpt()`; text correctly read through artoo is always valid UTF-8, so this only fires on bytes that entered the frame through a mis-declared source encoding.
created	*Creation timestamp.* `<POSIXct(1)> | NULL`. `NULL` (default) stamps the current time into `datasetJSONCreationDateTime`; freeze it for byte-stable output.
strict	*Suppress the* `_artoo` *extension block.* `<logical(1)>: default FALSE`. By default the file carries a single namespaced `_artoo` object when (and only when) there is content the closed Dataset-JSON vocabulary cannot express: SAS special-missing tags (`.A-.Z`, `._`), the recorded source encoding, and the per-variable informats, origins, codelist references, and significant digits. The `columns` array itself always uses only schema-legal keys. Data values stay plain `null`s either way, so a foreign reader degrades gracefully. **Note:** `strict = TRUE` writes a pure closed-vocabulary file that validates against the official CDISC schema, and warns (`artoo_warning_codec`) naming exactly what was dropped; those attributes will not survive a read-back.

Details

Full metadata, no loss. Unlike `.xpt`, a `.json` file records the complete `artoo_meta`: keySequence, codelist, origin, targetDataType, and significantDigits all survive. Dates, datetimes, and times are exchanged as ISO 8601 strings, or as SAS-epoch numbers when their `targetDataType` is `"integer"` (the ADaM numeric-date convention); `decimal` rides as a string so exact precision is preserved. The file is always UTF-8 (RFC 8259 / CDISC v1.1). `NaN` and infinite values are not valid CDISC numerics and abort the write.

Streaming write, whole-file read. The writer streams the `rows` array in bounded slabs (a `.json.gz` path gzips the stream transparently), but `read_json()` must parse the whole array at once. For multi-million-row datasets prefer the NDJSON variant (`write_ndjson()` / `read_ndjson()`), which bounds memory in both directions.

Value

The input `x`, invisibly, so a write can sit mid-pipeline.

See Also

`read_json()` for the inverse; `write_dataset()` for the generic dispatcher.

Examples

```
# ---- Example 1: write a conformed dataset as Dataset-JSON ----
#
# apply_spec() attaches the metadata; write_json() serializes the full
# itemGroup plus the data rows.
adsl <- apply_spec(cdisc_adsl, adam_spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".json")
write_json(adsl, path)

# ---- Example 2: a frozen timestamp for reproducible bytes ----
#
# Fixing `created` makes two writes byte-identical; the columns() pane on
# the written file shows the full metadata the file carries (DM is SDTM,
# so it conforms against the bundled sdtm_spec).
dm <- apply_spec(cdisc_dm, sdtm_spec, "DM", conformance = "off")
path2 <- tempfile(fileext = ".json")
write_json(dm, path2, created = as.POSIXct("2020-01-01", tz = "UTC"))
columns(path2)
```

`write_ndjson` *Write a dataset to CDISC Dataset-JSON NDJSON*

Description

Serialize a data frame to the newline-delimited variant of CDISC Dataset-JSON v1.1 (`.ndjson`): line 1 carries the complete metadata block, every following line one row array. The streaming end of the artoo workflow (spec -> apply_spec -> write_ndjson) for datasets too large for the array-form `.json` file; a thin wrapper over `write_dataset()` with `format = "ndjson"`.

Usage

```
write_ndjson(
  x,
  path,
  on_invalid = c("error", "translit", "fold", "replace", "ignore"),
  created = NULL,
  strict = FALSE
)
```

Arguments

`x`	*The dataset to write.* `<data.frame>: required`. Typically the output of `apply_spec()`, carrying `artoo_meta`.
`path`	*Destination* `.ndjson` *path.* `<character(1)>: required`. A `.ndjson.gz` path writes gzip-compressed bytes.
`on_invalid`	*Policy for values that are not valid UTF-8.* `<character(1)>: default "error"`. One of `"error"` (abort with `artoo_error_codec`), `"replace"` (substitute ? and warn with `artoo_warning_encoding`), `"ignore"` (drop the invalid bytes), or `"translit"` / `"fold"` (accepted for pipeline symmetry; behave as `"error"` here, since a byte-level invalidity has no character fold). See `write_json()` for when this fires.
`created`	*Creation timestamp.* `<POSIXct(1)> | NULL`. `NULL` (default) stamps the current time into `datasetJSONCreationDateTime`; freeze it for byte-stable output.
`strict`	*Suppress the* `_artoo` *extension block.* `<logical(1)>: default FALSE`. See `write_json()`: the same extension semantics apply to the metadata line.

Details

Bounded memory, both directions. The writer streams slabs of per-column JSON literals and `read_ndjson()` parses slab-sized line batches, so a multi-million-row dataset never materializes a whole `rows` array the way the `.json` codec must. A `.ndjson.gz` path gzips the stream transparently.

Value

The input `x`, invisibly, so a write can sit mid-pipeline.

See Also

`read_ndjson()` for the inverse; `write_json()` for the array-form file; `write_dataset()` for the generic dispatcher.

Examples

```
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)

# ---- Example 1: write a conformed dataset as NDJSON ----
#
# apply_spec() attaches the metadata; write_ndjson() streams the metadata
# line and one row per line.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".ndjson")
write_ndjson(adsl, path)
readLines(path, n = 2)[2]

# ---- Example 2: gzip the stream via the file extension ----
#
# A .ndjson.gz path compresses transparently; read_ndjson() inflates it.
gz <- tempfile(fileext = ".ndjson.gz")
write_ndjson(adsl, gz)
nrow(read_ndjson(gz))
```

`write_parquet` *Write a dataset to Apache Parquet*

Description

Serialize a data frame to an Apache Parquet (`.parquet`) file, storing the data natively while preserving the full `artoo_meta` as a CDISC-shaped sidecar in the file's key-value metadata. The emit end of the artoo workflow (spec -> apply_spec -> write_parquet); a thin wrapper over `write_dataset()` with `format = "parquet"`. Requires the lightweight `nanoparquet` package.

Usage

```
write_parquet(
  x,
  path,
  encoding = NULL,
  on_invalid = c("error", "translit", "fold", "replace", "ignore"),
  compression = "snappy"
)
```

Arguments

`x`	*The dataset to write.* `<data.frame>: required`. Typically the output of `apply_spec()`, carrying `artoo_meta`.
`path`	*Destination* `.parquet` *path.* `<character(1)>: required`.

encoding — *Source charset to record.* <character(1)> | NULL. The parquet bytes are always written as UTF-8 (the format's STRING type is UTF-8 by spec); encoding only records the data's original charset in the artoo_meta, so a later write_xpt() can reproduce the source bytes. NULL (default) leaves the recorded encoding untouched.

Tip: any SAS or IANA spelling listed by artoo_encodings() is accepted.

on_invalid — *Policy for values that are not valid UTF-8.* <character(1)>: default "error". One of "error" (abort with artoo_error_codec), "replace" (substitute ? and warn with artoo_warning_encoding), "ignore" (drop the invalid bytes), or "translit" / "fold" (accepted for pipeline symmetry; behave as "error" here, since a byte-level invalidity has no character fold). See write_json() for when this fires; parquet STRING bytes are UTF-8 by spec, exactly like Dataset-JSON.

compression — *Column compression codec.* <character(1)>: default "snappy". One of:

- "snappy" (default) — fast, the parquet ecosystem default.
- "gzip" — smaller files, slower.
- "zstd" — the best size/speed trade-off where supported.
- "uncompressed" — raw pages.

Details

Metadata where plain Parquet has none. A bare nanoparquet/arrow file drops labels, formats, and codelists; write_parquet() embeds the complete artoo_meta as a single Dataset-JSON-shaped string under the metadata_json key, so read_parquet() restores every CDISC attribute. The same string is what a .json file or an rds carries, so conversion between any two formats stays lossless. A reader without artoo still opens the data and can see the metadata_json block.

Value

The input x, invisibly, so a write can sit mid-pipeline.

See Also

read_parquet() for the inverse; write_dataset() for the generic dispatcher.

Examples

```
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)

# ---- Example 1: write a conformed dataset to Parquet ----
#
# apply_spec() attaches the metadata; write_parquet() stores the data
# natively and the metadata as a CDISC-shaped sidecar.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".parquet")
write_parquet(adsl, path)

# ---- Example 2: round-trip and confirm the metadata survived ----
#
```

```
# Reading it back yields an identical artoo_meta.
back <- read_parquet(path)
identical(get_meta(back)@columns, get_meta(adsl)@columns)
```

`write_rds` *Write a dataset to rds*

Description

Write a data frame to an R `.rds` file, preserving its `artoo_meta`. A thin wrapper over `write_dataset()` with `format = "rds"`; the rds carries the metadata both as live R attributes and as the language-agnostic `metadata_json` string, so `read_rds()` restores it exactly.

Usage

```
write_rds(x, path, encoding = NULL)
```

Arguments

`x`	*The dataset to write.* `<data.frame>: required`.
`path`	*Destination* `.rds` *path.* `<character(1)>: required`.
`encoding`	*Source charset to record.* `<character(1)> | NULL`. rds is R-native and faithful: strings are saved as-is, never transcoded. `encoding` only records the data's original charset in the `artoo_meta`, so a later `write_xpt()` can reproduce the source bytes. `NULL` (default) leaves the recorded encoding untouched. **Tip:** any SAS or IANA spelling listed by `artoo_encodings()` is accepted.

Value

The input `x`, invisibly, so a write can sit mid-pipeline.

See Also

`read_rds()` for the inverse; `write_dataset()` for the generic dispatcher.

Examples

```
spec <- artoo_spec(cdisc_adam_datasets, cdisc_adam_variables, codelists = cdisc_codelists)

# ---- Example 1: write a conformed dataset to rds ----
#
# apply_spec() attaches the metadata; write_rds() carries it into the file.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".rds")
write_rds(adsl, path)

# ---- Example 2: round-trip and confirm the metadata survived ----
```

```
#
# Reading it back yields an identical artoo_meta.
back <- read_rds(path)
identical(get_meta(back)@columns, get_meta(adsl)@columns)
```

`write_spec` *Write a specification to JSON, an Excel workbook, or Define-XML*

Description

Serialise a `artoo_spec`, dispatching on the file extension: a `.json` path writes artoo's native, lossless JSON; a `.xlsx` path writes a Pinnacle 21 (P21) style Excel workbook; a `.xml` path writes a submission-grade Define-XML 2.1 or 2.0 document. Each is the inverse of `read_spec()` on its format, which makes the spec converters free compositions: `read_spec("spec.xlsx") |> write_spec("define.xml")` turns a workbook into a define.xml in one line.

Usage

```
write_spec(spec, path, ...)
```

Arguments

`spec`	*The specification to serialise.* `<artoo_spec>: required`. Build one with `artoo_spec()` or `read_spec()`.
`path`	*Destination file.* `<character(1)>: required`. The extension picks the format: `.json` (native, lossless), `.xlsx` (P21 interchange; needs the `writexl` package), or `.xml` (Define-XML; needs the `xml2` package). Any other extension aborts with `artoo_error_input`.
`...`	*Format-specific options.* Ignored by the JSON and xlsx paths. Define-XML accepts: • `version` – `<character(1)> | NULL`. `"2.1"` (default) or `"2.0"`, resolved from the spec's own `define_version` when unset. Writing 2.0 from a 2.1-shaped spec warns about each construct 2.0 cannot carry. • `created` – the `CreationDateTime` stamp, formatted as UTC. Freeze it for a reproducible submission build; the default is the current time. • `stylesheet` – `<logical(1)> | <character(1)>: default TRUE`. `TRUE` writes the `xml-stylesheet` processing instruction and copies the bundled CDISC stylesheet beside the output; a string names a stylesheet without copying one; `FALSE` writes neither. • `data` – `<list of <data.frame>> | <character(1)> | NULL`. The datasets the define describes: a list named for each dataset, or one path to the folder holding them. artoo reads them and fills what the spec leaves blank; see **Data-aware writing**. • `data_format` – `<character> | NULL`. When `data` is a folder, restrict it to these formats, named as `artoo_formats()` lists them. Use it when a folder holds one dataset in two formats.

- `html` – `<logical(1)> | <character(1)>: default FALSE`. `TRUE` also renders the document through its stylesheet into a sibling `.html`; a string renders it to that path. Needs the `xslt` and `callr` packages.
- `validate` – `<logical(1)>: default TRUE`. Schema-validate before the file is put in place.

Details

Native JSON is the lossless format. Each slot is written as an array of row objects, with `NA` encoded as JSON `null` and numbers at full precision, so `read_spec()` rebuilds an identical `artoo_spec` through `artoo_spec()`. Object keys are emitted in a fixed order, so writing the same spec twice yields byte-identical output.

P21 xlsx is the interchange format. Sheets are emitted with the headers the P21 reader recognises (Define, Datasets, Variables, ValueLevel, Codelists, Methods, Comments, Documents; empty optional sheets are omitted), foreign keys repeated on every row (no merged cells), and the spec's `spec_standard()` as the Datasets sheet's `Standard` column. The study row writes back as the Define sheet's Attribute/Value pairs (`StudyName`, `StudyDescription`, `ProtocolName`). The `Data Type` column is written in the Define-XML / ODM vocabulary the workbook expects: a character variable is `text` (not the Dataset-JSON `string`), and `decimal` / `double` collapse to `float`, `boolean` / `URI` to `text`.

Columns the P21 vocabulary does not model are not lost: a foreign column carried on a slot is re-emitted verbatim under its own header, so an xlsx round-trip keeps user columns.

Note: the xlsx writer emits every sheet the spec has content for, so `standards`, `where_clauses`, `arm_displays`, `arm_results` and `dictionaries` survive a round trip. `method_expressions` does not survive whole: a workbook gives each method one row with one code cell, so a method carrying several formal expressions keeps only the first and the write says which methods it truncated. Write JSON when a spec has multi-expression methods, or when you need back the OIDs a Define-XML document chose.

Define-XML is the submission format. The `.xml` path emits Define-XML 2.1 or 2.0 (needs the `xml2` package) and SCHEMA-VALIDATES what it built before the file reaches its destination, so an invalid document never overwrites a good one. Value-level metadata is emitted whole: the parent variable's `def:ValueListRef`, the `def:ValueListDef`, a real `ItemDef` per value-level row, and the `def:WhereClauseRef` and `def:WhereClauseDef` that say which rows it applies to.

Identifiers already on the spec are reused verbatim, so a document read and written back keeps every OID a reviewer may have bookmarked; the rest are minted readably (`IG.DM`, `IT.DM.USUBJID`, `VL.VS.VSORRES`). Two schema-required attributes are derived rather than demanded: `def:Structure` falls back to the dataset keys, and `Purpose` follows the CDISC standard. A dataset with neither a structure nor keys aborts with `artoo_error_define` rather than being given an invented one.

HTML. A define.xml renders through an XSLT stylesheet, and the written document names one in a processing instruction and gets a copy of it alongside (a reference to a stylesheet that is not there is itself a conformance finding). An existing stylesheet beside the output is never overwritten, so a customised rendering survives. Pass `html = TRUE` to also materialise the rendered HTML: browsers are removing XSLT support, and a reviewer working from a submission archive should not need one.

What a folder measures. A named list describes the frames in memory; a folder describes the bytes on disk. Where the two differ the folder is right about what will be submitted – a transport file

pads to fixed width, so a value's trailing blanks are part of it there and not in R. A gzipped `.json` or `.ndjson` is matched like any other file; `.xpt.gz` is not, because `read_dataset()` does not read one either.

A folder instead of a list. `data` also takes one path to the folder holding the datasets. Each dataset the spec names is matched to a file whose basename is that name, ignoring case: `DM` to `dm.xpt`, `dm.json`, `dm.parquet`. The folder is inventoried, not descended. A dataset with no file is normal and reported, not an error, and a file the spec does not name is left alone. A dataset matching MORE than one file aborts rather than choosing: two formats can disagree about byte width, so picking one silently would change the document. Name the format with `data_format` to resolve it. A file that cannot be read aborts too, naming every unreadable file at once rather than stopping at the first.

Data-aware writing. Pass `data` and artoo reads the datasets the define describes, which a spec-only tool cannot. A blank `length` is filled from the real maximum byte width; a stated one shorter than the data is widened, because a length below the real maximum is a conformance finding, and the write says which variables it widened. A stated length LONGER than the data is left alone: a length is a claim about the domain, not about one extract. Value-level metadata is derived for the standard findings shapes – a result keyed by its test code, `TSVAL` by `TSPARMCD`, `QVAL` by `QNAM`, `AVAL` and `AVALC` by `PARAMCD` – with each derived row carrying the type and width of the rows it covers. A variable the spec already gives value-level rows to is never touched. A dataset with no records is flagged `def:HasNoData` when it also carries a comment explaining the absence, and left unflagged with a warning when it does not.

Fields with no P21 column (`itemoid`, `target_data_type`, per-variable `key_sequence`) likewise do not survive an xlsx round-trip; persist to JSON when you need the spec back exactly. The `Data Type` re-encoding is also non-injective: `decimal`, `double`, `boolean`, and `URI` fold to `float` or `text` on a read-back. A Define-XML `partialDate` / `partialDatetime` (and the other partial / incomplete subtypes) is read as the base `date` / `datetime` – CDISC Dataset-JSON v1.1 has no partial dataType – so it is written back as the base type.

Value

The output `path`*, invisibly.* Read it back with `read_spec()`.

See Also

Inverse: `read_spec()` reads native JSON, a P21 Excel workbook, or Define-XML back into a `artoo_spec`.

Check the Define-XML written: `validate_define()` schema-validates it, `lint_define()` checks its reference integrity.

Build / inspect: `artoo_spec()`, `spec_datasets()`, `spec_variables()`, `spec_standard()`.

Examples

```
# ---- Example 1: persist a spec to JSON, then read it back ----
#
# Build a spec from the bundled CDISC-pilot tables, write it to a temp
# JSON file, and confirm read_spec() reconstructs it intact.
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists
```

```
)
path <- tempfile(fileext = ".json")
write_spec(spec, path)
identical(read_spec(path), spec)

# ---- Example 2: the same spec as a P21 workbook ----
#
# The .xlsx path emits P21-shaped sheets; reading the workbook back
# recovers the P21-representable surface (here: the dataset names).
if (requireNamespace("writexl", quietly = TRUE)) {
  xlsx <- tempfile(fileext = ".xlsx")
  write_spec(spec, xlsx)
  spec_datasets(read_spec(xlsx))
}

# ---- Example 3: the same spec as a submission-grade define.xml ----
#
# Read the bundled minimal Define-XML, write it back with a frozen
# timestamp, and confirm the result is schema-valid. The write itself
# validates, so reaching this line already proves it.
if (requireNamespace("xml2", quietly = TRUE)) {
  dm <- read_spec(
    system.file("extdata", "define-minimal.xml", package = "artoo")
  )
  xml <- file.path(tempdir(), "define.xml")
  write_spec(dm, xml, created = "2020-01-01 00:00:00")
  validate_define(xml)
}

# ---- Example 4: let a folder of datasets fill the blanks ----
#
# A spec states a variable's length or leaves it blank. Point `data` at
# the folder holding the datasets and artoo matches each one the spec
# names to a file called after it, then fills the blanks from the real
# maximum byte width. It reports which files it used, because a define is
# a submission document and that is what a reader cannot recover from it.
if (requireNamespace("xml2", quietly = TRUE)) {
  folder <- file.path(tempdir(), "datasets")
  dir.create(folder, showWarnings = FALSE)
  write_json(
    apply_spec(cdisc_adsl, adam_spec, "ADSL", conformance = "off"),
    file.path(folder, "adsl.json")
  )
  from_folder <- file.path(tempdir(), "from-folder.xml")
  write_spec(adam_spec, from_folder, data = folder)
  lint_define(from_folder)
}
```

`write_template` *Write a blank Pinnacle 21 workbook to fill in*

Description

Emit an empty Excel workbook with the sheets and column headers `read_spec()` recognises, so a specification can be authored from the shape the reader wants rather than guessed at. Fill it in, read it back with `read_spec()`, and write a define.xml with `write_spec()`.

Usage

```
write_template(path, version = "2.1")
```

Arguments

`path`	*Destination workbook.* `<character(1)>: required`. An `.xlsx` path. Needs the `writexl` package.
`version`	*Define-XML version the workbook is for.* `<character(1)>: default "2.1"`.

- `"2.1"` (default)
- `"2.0"` – omits the columns 2.1 introduced.

Details

The headers are the reader's own. Every sheet name and every column comes from the same maps `read_spec()` matches against, so a template cannot offer a column that would be silently ignored, and a column added to the reader appears here without a second list to maintain.

A 2.0 template is narrower. Columns Define-XML 2.1 introduced (`SubClass` and `Has No Data` on Datasets, `Source` and `Has No Data` on Variables) are omitted from a 2.0 template, because a column no 2.0 document can carry is one an author fills for nothing.

Only Datasets and Variables are required. Every other sheet may be left empty; `read_spec()` omits what it finds nothing in.

Conditions live on the `WhereClauses` sheet. A value-level row names a condition by its `ID` and the `WhereClauses` sheet defines it, one row per comparison, rows sharing an `ID` being ANDed together. This is the shape the widest range of tools import. `read_spec()` also reads the newer shape, which drops that sheet and writes the condition into the `Where Clause` cell as an expression.

`Standards` **is an artoo extension.** Define-XML 2.1 carries a `def:Standards` block that the workbook format has no sheet for, so artoo offers one rather than lose it.

Value

The output `path`*, invisibly.* Fill it in, then read it with `read_spec()`.

See Also

Fill it in: `read_spec()` reads the completed workbook.

Then write: `write_spec()` turns the spec into a define.xml, `validate_define()` and `lint_define()` check the result.

Examples

```
# ---- Example 1: a blank workbook, and what it offers ----
#
# The template carries every sheet read_spec() recognises. Datasets and
# Variables are the two it requires; the rest may be left empty.
path <- tempfile(fileext = ".xlsx")
write_template(path)
if (requireNamespace("readxl", quietly = TRUE)) {
  readxl::excel_sheets(path)
}

# ---- Example 2: the 2.0 template omits what 2.0 cannot carry ----
#
# `Source` records who collected a value, which Define-XML 2.1 added. A
# 2.0 workbook has no column for it, so the template does not offer one.
old <- tempfile(fileext = ".xlsx")
write_template(old, version = "2.0")
if (requireNamespace("readxl", quietly = TRUE)) {
  setdiff(
    names(readxl::read_excel(path, sheet = "Variables")),
    names(readxl::read_excel(old, sheet = "Variables"))
  )
}
```

`write_xpt` *Write a dataset to SAS XPORT*

Description

Serialize a data frame to a SAS Transport (`.xpt`) file in v5 (the FDA submission standard) or v8 (extended names and labels), preserving the `artoo_meta` a column can hold. The emit end of the artoo workflow (spec -> apply_spec -> write_xpt); a thin wrapper over `write_dataset()` with `format = "xpt"`.

Usage

```
write_xpt(
  x,
  path,
  version = 5,
  encoding = NULL,
  on_invalid = c("error", "translit", "fold", "replace", "ignore"),
  created = NULL
)
```

Arguments

x	*The dataset to write.* <data.frame>: required. Typically the output of apply_spec(), carrying artoo_meta.
path	*Destination* .xpt *path.* <character(1)>: required.
version	*XPORT transport version.* <integer(1)>: default 5. 5 (the FDA standard: names <= 8 characters, labels <= 40 bytes) or 8 (names <= 32, long labels).
encoding	*Target charset.* <character(1)> | NULL. NULL (default) inherits the source encoding recorded in artoo_meta, else UTF-8. IANA and SAS names ("US-ASCII", "wlatin1") both work. **Tip:** any SAS or IANA spelling listed by artoo_encodings() is accepted.
on_invalid	*Policy for values not representable in* encoding. <character(1)>: default "error". The same policy vocabulary as the UTF-8 writers (write_json(), write_ndjson(), write_parquet()): • "error" (default) — abort with artoo_error_codec, naming the offenders. • "translit" — fold smart punctuation (curly quotes, en/em dashes, ellipsis, bullet) to its exact ASCII form per the SAS NLS punctuation table and warn; a character with no fold (a diacritic) still aborts. • "fold" — "translit" plus the ICU Latin-ASCII accent strip (Ö to O, ß to ss, Æ to AE), and warn. Lossy on names — the original characters are not recoverable; a character neither table maps (the Euro sign) still aborts. • "replace" — substitute one ? per unrepresentable character and warn with artoo_warning_encoding. • "ignore" — drop the unrepresentable characters silently. **Tip:** for a US-ASCII submission write, "translit" fixes the word-processor punctuation that dominates real findings while keeping genuine data corruption loud; reach for "fold" only when accent stripping is an accepted, documented step of the migration.
created	*Header timestamp.* <POSIXct(1)> | NULL. NULL (default) stamps the current time; freeze it for byte-stable output.

Details

What XPORT can carry. An .xpt file's NAMESTR stores only variable name, label, length, and SAS format. CDISC metadata beyond that (keySequence, codelist, origin, targetDataType, ...) and the source encoding are not representable in the bytes; they ride the in-session artoo_meta and the sidecar in self-describing formats (Dataset-JSON, Parquet, rds). XPORT also cannot distinguish an empty string from NA (both store as blanks) and drops trailing spaces.

Character ISO dates (-DTC) write as text. A character column whose dataType is date/datetime/time with no numeric targetDataType is the CDISC ISO 8601 text form — the SDTM --DTC convention — and stores as a character variable, partial dates ("1951", "1951-12") included, byte for byte. The SAS-numeric encoding (with DATE9.-style formats) is used for columns that are R Date/POSIXct/hms or whose metadata records targetDataType = "integer" (the ADaM numeric-date convention). A character column *under* targetDataType = "integer" aborts loudly — a partial date can never become a SAS numeric silently.

Value

The input x, invisibly, so a write can sit mid-pipeline.

See Also

read_xpt() for the inverse; write_dataset() for the generic dispatcher.

Examples

```
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists
)

# ---- Example 1: write a conformed dataset as v5 (FDA standard) ----
#
# apply_spec() attaches the metadata; write_xpt() carries the label, length,
# and SAS format for each variable into the transport file.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
path <- tempfile(fileext = ".xpt")
write_xpt(adsl, path)

# ---- Example 2: v8 for long names, with a frozen timestamp ----
#
# Version 8 keeps names over 8 characters; a fixed `created` makes the bytes
# reproducible. Reading it back shows the labels, types, and record count
# survived the transport. DM is SDTM, so it conforms against the bundled
# sdtm_spec.
dm <- apply_spec(cdisc_dm, sdtm_spec, "DM", conformance = "off")
path8 <- tempfile(fileext = ".xpt")
write_xpt(dm, path8, version = 8, created = as.POSIXct("2020-01-01", tz = "UTC"))
get_meta(read_xpt(path8))@dataset$records
```

---

xpt_members *List the members of a SAS XPORT transport file*

Description

Report every dataset (member) a SAS Transport (.xpt) file holds, with its label, variable count, and row count — the survey step before read_xpt() with member = picks one. A single-member file (the FDA submission convention) returns one row.

Usage

```
xpt_members(path)
```

Arguments

path	*Source* .xpt *path.* <character(1)>: required. A file that is not a valid XPORT library aborts with artoo_error_codec.

Details

v5 has no recorded row count. A v8 member records its rows; a v5 member's count is derived from the byte span up to the next member (or end of file) minus trailing padding, so an all-character v5 member whose last row is entirely blank reports one row fewer (the documented v5 ambiguity, see write_xpt()).

Value

A <data.frame> with one row per member and columns member (1-based index), name, label, nvars, and nobs. Pass member or name to read_xpt().

See Also

read_xpt() with member = to read one of them.

Examples

```
spec <- artoo_spec(
  cdisc_adam_datasets, cdisc_adam_variables,
  codelists = cdisc_codelists
)

# ---- Example 1: a single-member file reports one row ----
#
# The FDA convention is one dataset per transport file.
dm <- apply_spec(cdisc_dm, sdtm_spec, "DM", conformance = "off")
p <- tempfile(fileext = ".xpt")
write_xpt(dm, p)
xpt_members(p)

# ---- Example 2: survey a multi-member file, then read one member ----
#
# Concatenate two single-member files into one library and list it.
adsl <- apply_spec(cdisc_adsl, spec, "ADSL", conformance = "off")
p2 <- tempfile(fileext = ".xpt")
write_xpt(adsl, p2)
multi <- tempfile(fileext = ".xpt")
writeBin(
  c(
    readBin(p, "raw", file.size(p)),
    readBin(p2, "raw", file.size(p2))[-(1:240)]
  ),
  multi
)
xpt_members(multi)
```

Index

∗ **datasets**
cdisc_adae, 14
cdisc_adsl, 14
cdisc_dm, 15
cdisc_spec, 15
cdisc_specs, 16
cdisc_suppdm, 17
cdisc_ts, 18
cdisc_vs, 18

adam_spec *(*cdisc_specs*)*, 16
apply_spec, 3
apply_spec(), *8*, *19, 20*, *22–28*, *30*, *45–47*, *58*, *62, 63*, *65, 66*, *75*
artoo_checks, 6
artoo_checks(), *19–21*, *24*, *28, 29*
artoo_encodings, 9
artoo_encodings(), *36*, *38, 39*, *42*, *67, 68*, *75*
artoo_formats, 10
artoo_formats(), *9*, *32*, *34*, *62*, *69*
artoo_spec, 11
artoo_spec(), *15*, *17*, *30*, *41*, *54, 55*, *59*, *61*, *69–71*

cdisc_adae, 14
cdisc_adam_datasets *(*cdisc_spec*)*, 15
cdisc_adam_variables *(*cdisc_spec*)*, 15
cdisc_adsl, 14
cdisc_codelists *(*cdisc_spec*)*, 15
cdisc_dm, 15
cdisc_sdtm_datasets *(*cdisc_spec*)*, 15
cdisc_sdtm_variables *(*cdisc_spec*)*, 15
cdisc_spec, 15
cdisc_specs, 16
cdisc_suppdm, 17
cdisc_ts, 18
cdisc_vs, 18
check_spec, 19
check_spec(), *4*, *6*, *8*, *20, 21*, *24*, *26*, *28*, *44*, *47, 48*, *56*
check_study, 20
check_study(), *20*, *44*
columns, 22
columns(), *33*
conformance, 23
conformance(), *4–6*

decode_column, 25
decode_column(), *5, 6*

get_meta, 27
get_meta(), *6*, *23*, *29*, *34*, *36–38*, *43*, *45, 46*, *58*

is_artoo_checks, 28
is_artoo_meta, 29
is_artoo_meta(), *30*
is_artoo_spec, 30
is_artoo_spec(), *13*

lint_define, 31
lint_define(), *59*, *71*, *73*

members, 32

read_dataset, 34
read_dataset(), *10*, *23*, *32, 33*, *35–40*, *42, 43*, *62*, *71*
read_json, 35
read_json(), *9*, *37*, *64*
read_ndjson, 36
read_ndjson(), *64, 65*
read_parquet, 38
read_parquet(), *67*
read_rds, 39
read_rds(), *34, 35*, *68*
read_spec, 40
read_spec(), *59*, *69–71*, *73*
read_xpt, 42
read_xpt(), *9*, *34*, *76, 77*
repair_spec, 44

repair_spec(), *5, 6*, *20, 21*, *47*

sdtm_spec *(*cdisc_specs*)*, 16
set_meta, 45
set_meta(), *6*, *27–29*, *58*
set_type, 46
set_type(), *4–6*, *44*
spec_codelists, 48
spec_codelists(), *13*, *26*, *57*
spec_comments, 49
spec_comments(), *51*, *53*, *61*
spec_datasets, 50
spec_datasets(), *13*, *17*, *19*, *41*, *52*, *55–57*, *71*
spec_documents, 51
spec_documents(), *49*, *53*
spec_keys, 52
spec_keys(), *13*, *50*
spec_methods, 53
spec_methods(), *49*, *51*, *61*
spec_standard, 54
spec_standard(), *16, 17*, *55*, *70, 71*
spec_study, 55
spec_study(), *13*, *54*
spec_variables, 56
spec_variables(), *13*, *17*, *41*, *48*, *50*, *52*, *71*
sync_meta, 57

validate_define, 58
validate_define(), *31*, *71*, *73*
validate_spec, 60
validate_spec(), *12, 13*, *19–21*, *41*, *47*, *49*, *51*, *53*

write_dataset, 61
write_dataset(), *10*, *34, 35*, *63–68*, *74*, *76*
write_json, 63
write_json(), *36*, *65*, *67*, *75*
write_ndjson, 64
write_ndjson(), *37*, *64*, *75*
write_parquet, 66
write_parquet(), *38*, *75*
write_rds, 68
write_rds(), *39, 40*, *62*
write_spec, 69
write_spec(), *16*, *40, 41*, *44*, *47*, *73*
write_template, 72
write_xpt, 74
write_xpt(), *9*, *43*, *62, 63*, *67, 68*, *77*

xpt_members, 76
xpt_members(), *23*, *32, 33*, *43*