Package 'biocohort'

September 18, 2026

Title Cohort Objects for Subjects and Samples in Omics Studies

Version 0.1.1

Description Keeps the subjects, samples, and analysis outputs of a study in one validated object. It starts from a sample manifest with one row per sample, which is read, checked, and split into a subject table and a sample map. Species and assay are plain values in those tables rather than fixed types, so the same object serves any organism and any omics assay. From that object the package writes the sample sheet a pipeline expects, pairs tumor and normal samples on demand, and records where each analysis writes its output so the files can be loaded back in by subject or by pair. Manual corrections are kept in an audit trail. Results can also be translated across genome builds or species, with liftover for coordinates and ortholog mapping for genes.

License MIT + file LICENSE

URL https://www.samuelbharti.com/biocohort/, https://github.com/samuelbharti/biocohort

BugReports https://github.com/samuelbharti/biocohort/issues

Encoding UTF-8

Language en-US

Depends R (>= 4.1)

Imports S7, cli, rlang, checkmate, fs, readr, dplyr (>= 1.1.0), tibble

Suggests testthat (>= 3.0.0), pkgdown, knitr, rmarkdown, rtracklayer, GenomicRanges, IRanges, S4Vectors, babelgene, withr, readxl, writexl, arrow, SummarizedExperiment, SeuratObject, yaml

SystemRequirements CrossMap (optional, for the CrossMap liftover backend)

VignetteBuilder knitr

Config/testthat/edition 3

LazyData true

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Samuel Bharti [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0003-4190-7058>),
Barret Schloerke [ths] (ORCID: <https://orcid.org/0000-0001-9986-114X>),
Carson Sievert [ths] (ORCID: <https://orcid.org/0000-0002-4958-2844>),
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Samuel Bharti <samuelbharti.io@gmail.com>

Repository CRAN

Date/Publication 2026-09-18 11:20:14 UTC

Contents

biocohort-package . . . 3
AnalysisSpec . . . 6
analysis_files . . . 8
analysis_list . . . 9
analysis_register . . . 10
analysis_spec . . . 12
analysis_spec_new . . . 13
apply_corrections . . . 16
as_coldata . . . 17
check_paths . . . 18
Cohort . . . 19
cohort_contrasts . . . 21
cohort_derive . . . 22
cohort_filter . . . 23
cohort_groups . . . 25
cohort_new . . . 26
cohort_qc . . . 27
cohort_read . . . 29
cohort_save . . . 30
completeness . . . 30
corrections_log . . . 31
derive_log . . . 32
ensure_dir . . . 32
example_cohort . . . 33
join_metadata . . . 34
liftover_backends . . . 35
liftover_crossmap . . . 36
liftover_intervals . . . 37
liftover_rtracklayer . . . 39
liftover_vcf . . . 40
load_analyses . . . 41
load_analysis . . . 42
manifest_from_wide . . . 44
orthologize . . . 45

ortholog_babelgene . . . 46
ortholog_backends . . . 47
ortholog_genes . . . 48
project_path . . . 49
project_root . . . 50
qc_log . . . 51
read_corrections . . . 51
read_dotenv . . . 52
read_manifest . . . 53
read_manifest_csv . . . 54
read_study_yaml . . . 56
register_liftover_backend . . . 57
register_ortholog_backend . . . 58
samples . . . 59
sample_pairs . . . 60
sample_sheet . . . 62
sample_sheet_templates . . . 64
Study . . . 64
study_new . . . 66
Subject . . . 68
subject . . . 69
subjects . . . 70
subject_new . . . 71
translate . . . 72
TranslationResult . . . 74
translation_report . . . 76
translation_stats . . . 77
validate_cohort . . . 78
validate_manifest . . . 79
write_manifest . . . 81
write_study_yaml . . . 82

Index **84**

`biocohort-package` *biocohort: Subject and Sample Rosters for Omics Studies*

Description

biocohort keeps the subjects, samples, and analysis outputs of a study in one validated object, called a `Cohort`. Species and assays are values in the data, not columns or classes, so the same functions work for a rat exome study, a mouse single-cell study, a proteomics study, or any other organism and assay.

From a manifest to a cohort

- `read_manifest()` reads a manifest from CSV, TSV, or Excel and returns `subject_tbl` and `sample_map`. `manifest_from_wide()` reshapes a wide, one-row-per-subject table into the long form first.
- `validate_manifest()` does the actual checking: it coerces every column to character, fills in `role` and `species` where they are missing, and splits subject-level columns from sample-level ones.
- `cohort_new()` builds a `Cohort` from validated tables. `study_new()` attaches optional project metadata (title, aims, genome builds).

Reading a cohort

- `subjects()`, `samples()`, and `completeness()` return plain tibbles.
- `subject()` reads one subject as a `Subject` object.
- `cohort_filter()` keeps a subset of subjects or assays and returns a cohort that is still valid.
- `sample_pairs()` derives tumor and normal pairs from `sample_map` on demand, with configurable role labels.

Quality control, groups, and derived columns

- `cohort_qc()` flags or drops subjects or samples, with a required reason. `qc_log()` reads the audit trail every call appends to the cohort, which survives later `cohort_filter()` calls.
- `cohort_groups()` groups a cohort's subjects by one or more columns. `cohort_contrasts()` enumerates every pairwise contrast between those groups, ready to filter one side against the other.
- `cohort_derive()` bins an existing numeric column at named cutoffs and writes the result as a new column, so a cutoff is a value passed in, not code. `derive_log()` reads its provenance.

Writing files for other tools

- `sample_sheet()` writes the sample list a pipeline expects, with built-in templates for a few common nf-core pipelines.
- `check_paths()` tests that the file paths named in a cohort exist.
- `as_coldata()` and `join_metadata()` carry cohort metadata into a `SummarizedExperiment`, a Seurat object, or a plain data frame.
- `write_manifest()`, `cohort_save()`, and `cohort_read()` keep a cohort as a file in a project instead of a script that rebuilds it each time.

Analysis outputs

- `analysis_spec_new()` and `analysis_register()` describe where an analysis writes its output and how to read it back.
- `load_analysis()` and `load_analyses()` resolve the path for every subject or pair, read the files, and record which ones were found.

Cross-species translation

- `translate()` moves a feature table across genome builds or species. Coordinate features go through a liftover backend (`liftover_intervals()`). Gene features go through an ortholog backend (`ortholog_genes()`). Both kinds of backend are pluggable through `register_liftover_backend()` and `register_ortholog_backend()`.

Configuration

- `read_study_yaml()` builds a cohort from one YAML file that names the study, the manifest, the file paths, and the registered analyses. `write_study_yaml()` writes one back.
- `apply_corrections()` and `read_corrections()` apply documented overrides to a manifest and keep an audit trail.

Data tables

- `subject_tbl`: one row per subject. Always has `subject_id` and `species`, plus any other subject-level metadata (`sex`, `genotype`, `strain`, ...).
- `sample_map`: one row per sample, in long format. Always has `subject_id`, `assay`, `sample_id`, and `role`. A new assay is a new row, never a new column.
- `completeness_tbl`: one row per `subject_id` and `assay` pair, with the sample count.

Further reading

Three articles ship with the package. `vignette("biocohort", package = "biocohort")` walks through a manifest, a cohort, and a sample sheet end to end. `vignette("glossary", package = "biocohort")` defines the terms used across the package. `vignette("naming-conventions", package = "biocohort")` lists the standard names for columns, objects, and files.

Author(s)

Maintainer: Samuel Bharti <samuelbharti.io@gmail.com> (ORCID) [copyright holder]

Authors:

- Samuel Bharti <samuelbharti.io@gmail.com> (ORCID) [copyright holder]

Other contributors:

- Barret Schloerke <barret@posit.co> (ORCID) [thesis advisor]
- Carson Sievert <carson@posit.co> (ORCID) [thesis advisor]
- Posit Software, PBC (ROR) [copyright holder, funder]

See Also

Useful links:

- https://www.samuelbharti.com/biocohort/
- https://github.com/samuelbharti/biocohort
- Report bugs at https://github.com/samuelbharti/biocohort/issues

`AnalysisSpec` *S7 AnalysisSpec class*

Description

An immutable S7 class for defining analysis specifications in a Cohort. AnalysisSpec objects describe how to locate, read, and interpret analysis output files with templated paths and standardized readers.

Usage

```
AnalysisSpec(
  name = character(0),
  assay = character(0),
  level = character(0),
  format = NA_character_,
  description = NA_character_,
  path_template = NA_character_,
  root_key = NA_character_,
  reader = NA_character_,
  key_cols = character(0),
  feature_type = NA_character_,
  gene_col = NA_character_,
  id_type = NA_character_,
  tumor_role = "tumor",
  normal_role = "normal",
  pair_sep = "__"
)
```

Arguments

`name`	Character scalar for unique analysis name (key in registry).
`assay`	Character scalar for the assay label, spelled as in the cohort's `sample_map` (e.g., "wes", "wgs", "scrna"). Required.
`level`	Character scalar for the granularity at which the analysis produces results. Must be one of: • `"subject"`: one result per subject. • `"pair"`: one result per tumor/normal (case/control) pair, as derived by `sample_pairs()` from the cohort's `sample_map`. • `"cohort"`: a single result for the whole cohort. Required.
`format`	Character scalar for file format (e.g., "rds", "tsv", "txt"). Optional. `analysis_spec_new()` fills it from the `path_template` extension. NA when unknown.
`description`	Character scalar for human-readable description of the analysis. Optional, defaults to NA.

path_template — Character scalar for templated path to analysis output. Supports substitution tokens: {root} (from root_key), {subject_id}, and the pair tokens {tumor_sample_id}, {normal_sample_id}, {pair_id} (the latter three supplied by sample_pairs() for level = "pair"). Optional, defaults to NA.

root_key — Character scalar for key in cohort@paths list to use as the {root} template value (e.g., "msi_root", "sig_root", "wes_root"). Optional, defaults to NA.

reader — Character scalar for function name to read files matching this spec (e.g., "readr::read_tsv", "read.csv"). Optional. analysis_spec_new() fills it from format. NA when unknown.

key_cols — Character vector of column names that must be present in the loaded analysis table. load_analysis() checks them after reading. Optional. analysis_spec_new() fills it by level.

feature_type — Optional character scalar declaring how this analysis's features translate across species in translate(): "interval" (liftover) or "gene" (ortholog mapping). Optional, defaults to NA.

gene_col — Optional character scalar naming the gene-identifier column for feature_type = "gene". Optional, defaults to NA.

id_type — Optional gene identifier type for feature_type = "gene": "symbol", "entrez", or "ensembl". Optional, defaults to NA.

tumor_role — Character scalar naming the sample role on the tumor (or case) side of a pair. Used for level = "pair". Default "tumor".

normal_role — Character scalar naming the sample role on the normal (or control) side of a pair. Used for level = "pair". Default "normal".

pair_sep — Character scalar placed between the two sample ids when sample_pairs() builds pair_id. Used for level = "pair". Default "__".

Details

Use analysis_spec_new() to construct AnalysisSpec objects with immediate validation. AnalysisSpec objects are typically registered in a Cohort via analysis_register().

Access properties via the @ operator:

```
spec@name
spec@assay
spec@level
spec@format
spec@description
spec@path_template
spec@root_key
spec@reader
spec@key_cols
spec@tumor_role
spec@normal_role
spec@pair_sep
```

Value

An `AnalysisSpec` object with the given properties.

See Also

`analysis_spec_new()` for object construction, `analysis_register()` for registering specs in a Cohort

Examples

```
# The raw constructor. analysis_spec_new() fills format, reader, and
# key_cols in from the path template and the level; this does not.
spec <- AnalysisSpec(
  name = "somatic_vars", assay = "wes", level = "pair",
  format = "tsv", reader = "readr::read_tsv",
  key_cols = c("subject_id", "pair_id")
)
spec@level
spec@key_cols
```

---

analysis_files *Retrieve analysis file manifests from a cohort*

Description

After `load_analyses()`, returns the per-analysis file manifests (resolved paths and whether each existed) recorded during loading.

Usage

```
analysis_files(cohort)
```

Arguments

`cohort`	A Cohort produced by `load_analyses()`.

Value

A named list of tibbles, one per loaded analysis. Each has the unit keys, `path`, and `exists`. When the cohort has not been loaded, an empty tibble with columns `path` and `exists`.

See Also

`load_analyses()`

Examples

```
analysis_files(example_cohort)
```

`analysis_list` *List registered analysis specifications*

Description

Returns a tibble with one row per registered AnalysisSpec in a Cohort. Summarizes key metadata for quick inspection of available analyses.

Usage

```
analysis_list(cohort)
```

Arguments

`cohort`	A Cohort object.

Details

The returned tibble includes only the most essential metadata fields for discovery and filtering. Use `analysis_spec()` to retrieve the full AnalysisSpec object including description, path_template, and key_cols.

Value

A tibble with the following columns:

- `name`: Analysis name (chr)
- `assay`: Assay type (chr)
- `level`: Data level - "subject", "pair", or "cohort" (chr)
- `format`: File format (chr)
- `reader`: Reader function name (chr)
- `root_key`: Optional root path key (chr)

If the cohort has no registered specs, returns an empty tibble with these columns.

See Also

`analysis_register()` for registering specs, `analysis_spec()` for retrieving a full spec object

Examples

```
# Create and register specs
study <- study_new(study_id = "STUDY001", title = "My Study")
manifest <- data.frame(
  subject_id = c("S1", "S1", "S2", "S2"),
  species = c("rat", "rat", "rat", "rat"),
  assay = c("wes", "wes", "wes", "wes"),
```

```
  sample_id = c("WES_T1", "WES_N1", "WES_T2", "WES_N2"),
  role = c("tumor", "normal", "tumor", "normal")
)
parsed <- validate_manifest(manifest)
cohort <- cohort_new(
  subject_tbl = parsed$subject_tbl,
  sample_map = parsed$sample_map,
  study = study
)

spec1 <- analysis_spec_new(
  name = "somatic_vars",
  assay = "wes",
  level = "pair",
  format = "tsv",
  reader = "read_tsv",
  key_cols = c("pair_id")
)

spec2 <- analysis_spec_new(
  name = "gene_expr",
  assay = "scrna",
  level = "subject",
  format = "rds",
  reader = "readRDS",
  key_cols = c("subject_id")
)

cohort <- analysis_register(cohort, spec1)
cohort <- analysis_register(cohort, spec2)
analysis_list(cohort)
```

`analysis_register` *Register an analysis specification in a cohort*

Description

Registers an AnalysisSpec in a Cohort's registry, enabling standardized access and discovery of analysis specifications. Returns a new Cohort object with the spec added (immutable update pattern).

Usage

```
analysis_register(cohort, spec)
```

Arguments

`cohort`	A Cohort object to update.
`spec`	An AnalysisSpec object to register. The spec name is used as the registry key.

Details

Validates that:

- cohort is a Cohort object
- spec is an AnalysisSpec object

Value

A new Cohort object with the spec added to the registry. If a spec with the same name already exists, it is replaced. Note that this follows the immutable S7 pattern: the original cohort is not modified.

See Also

analysis_spec_new() for creating specs, analysis_list() for listing registered specs, analysis_spec() for retrieving a spec from registry

Examples

```
# Create a cohort
study <- study_new(study_id = "STUDY001", title = "My Study")
manifest <- data.frame(
  subject_id = c("S1", "S1", "S2", "S2"),
  species = c("rat", "rat", "rat", "rat"),
  assay = c("wes", "wes", "wes", "wes"),
  sample_id = c("WES_T1", "WES_N1", "WES_T2", "WES_N2"),
  role = c("tumor", "normal", "tumor", "normal")
)
parsed <- validate_manifest(manifest)
cohort <- cohort_new(
  subject_tbl = parsed$subject_tbl,
  sample_map = parsed$sample_map,
  study = study
)

# Create and register an analysis spec
spec <- analysis_spec_new(
  name = "somatic_vars",
  assay = "wes",
  level = "pair",
  format = "tsv",
  reader = "read_tsv",
  key_cols = c("pair_id")
)

cohort_with_spec <- analysis_register(cohort, spec)
print(analysis_list(cohort_with_spec))
```

`analysis_spec` *Retrieve an analysis specification from registry*

Description

Retrieves a fully-specified AnalysisSpec object from a Cohort's registry by name. Useful for accessing all properties of a registered analysis (description, path_template, key_cols, etc.).

Usage

```
analysis_spec(cohort, name)
```

Arguments

`cohort`	A Cohort object.
`name`	Character scalar with the name of the AnalysisSpec to retrieve.

Details

Raises an informative error if the spec name is not found in the registry.

Value

The AnalysisSpec object if found.

See Also

`analysis_register()` for registering specs, `analysis_list()` for listing all registered specs

Examples

```
# Create and register a spec
study <- study_new(study_id = "STUDY001", title = "My Study")
manifest <- data.frame(
  subject_id = c("S1", "S1", "S2", "S2"),
  species = c("rat", "rat", "rat", "rat"),
  assay = c("wes", "wes", "wes", "wes"),
  sample_id = c("WES_T1", "WES_N1", "WES_T2", "WES_N2"),
  role = c("tumor", "normal", "tumor", "normal")
)
parsed <- validate_manifest(manifest)
cohort <- cohort_new(
  subject_tbl = parsed$subject_tbl,
  sample_map = parsed$sample_map,
  study = study
)

spec <- analysis_spec_new(
  name = "somatic_vars",
```

```
  assay = "wes",
  level = "pair",
  format = "tsv",
  reader = "read_tsv",
  key_cols = c("pair_id")
)

cohort_with_spec <- analysis_register(cohort, spec)

# Retrieve the spec
retrieved_spec <- analysis_spec(cohort_with_spec, "somatic_vars")
print(retrieved_spec@reader)  # "read_tsv"
```

`analysis_spec_new` *Create an AnalysisSpec object*

Description

Constructs an AnalysisSpec object that defines how to locate, read, and interpret a specific analysis output. Validates all required fields and fills `format`, `reader`, and `key_cols` with defaults when they are not given.

Usage

```
analysis_spec_new(
  name,
  assay,
  level,
  format = NA_character_,
  description = NA_character_,
  path_template = NA_character_,
  root_key = NA_character_,
  reader = NA_character_,
  key_cols = NULL,
  feature_type = NA_character_,
  gene_col = NA_character_,
  id_type = NA_character_,
  tumor_role = "tumor",
  normal_role = "normal",
  pair_sep = "__"
)
```

Arguments

`name`	Character scalar for unique analysis name. Must be at least 1 character long. Serves as key in the cohort registry.

assay	Character scalar for the assay label, spelled as in the cohort's sample_map (e.g., "wes", "wgs", "scrna"). Required. Subject and pair units are enumerated from the samples with this assay.
level	Character scalar for the granularity at which the analysis produces results. Must be one of "subject" (one result per subject), "pair" (one result per tumor/normal pair, see sample_pairs()), or "cohort" (a single result for the whole cohort). Required.
format	Character scalar for file format (e.g., "rds", "tsv", "txt"). Optional. Defaults to the extension of path_template, or NA when there is no template.
description	Character scalar for human-readable description. Optional, defaults to NA.
path_template	Character scalar for templated file path. Supports tokens: {root} (from root_key), {subject_id}, and the pair tokens {tumor_sample_id}, {normal_sample_id}, {pair_id} (from sample_pairs()). Optional, defaults to NA.
root_key	Character scalar for key in cohort@paths to use as {root}. Optional, defaults to NA.
reader	Character scalar for reader function name (e.g., "readr::read_tsv", "read.csv"). Optional. Defaults by format: "csv" to "readr::read_csv", "tsv" and "txt" to "readr::read_tsv", "rds" to "readRDS", "parquet" to "arrow::read_parquet". NA for any other format. load_analysis() errors when neither the spec nor its reader argument names a reader.
key_cols	Character vector of column names that must be present in the loaded table. Optional. Defaults by level: "subject_id" for subject, c("subject_id", "pair_id") for pair, and none for cohort. Subject and pair specs need at least one key column.
feature_type	Optional character scalar declaring how this analysis's features are translated across species by translate(). One of "interval" (coordinate features, translated by liftover) or "gene" (gene-level features, translated by ortholog mapping). Defaults to NA (analysis is skipped by cohort-level translation).
gene_col	Optional character scalar naming the gene-identifier column, used when feature_type = "gene". Defaults to NA (treated as "gene").
id_type	Optional gene identifier type for feature_type = "gene": one of "symbol", "entrez", "ensembl". Defaults to NA (treated as "symbol").
tumor_role	Character scalar naming the sample role on the tumor (or case) side of a pair. Passed to sample_pairs() for level = "pair". Default "tumor".
normal_role	Character scalar naming the sample role on the normal (or control) side of a pair. Passed to sample_pairs() for level = "pair". Default "normal".
pair_sep	Character scalar placed between the two sample ids in pair_id. Passed to sample_pairs() for level = "pair". Default "__".

Details

This constructor validates that:

- name and assay are non-empty strings
- level is one of: "subject", "pair", "cohort"

- `format` and `reader`, if given, are non-empty strings
- `key_cols` is a character vector, non-empty for subject and pair specs
- `feature_type`, if given, is one of "interval" or "gene"
- `id_type`, if given, is one of "symbol", "entrez", "ensembl"
- `tumor_role`, `normal_role`, and `pair_sep` are non-empty strings

Value

An AnalysisSpec object with validated fields.

See Also

AnalysisSpec for class documentation, `analysis_register()` for registering in a Cohort

Examples

```
# A pair-level somatic variant spec. format, reader, and key_cols come
# from the template extension and the level.
spec_sv <- analysis_spec_new(
  name = "somatic_vars",
  assay = "wes",
  level = "pair",
  description = "Somatic variants in tumor-normal pairs",
  path_template = "{root}/somatic/{pair_id}.variants.tsv",
  root_key = "wes_root"
)
spec_sv@format
spec_sv@reader
spec_sv@key_cols

# A subject-level gene expression spec with an explicit reader.
spec_expr <- analysis_spec_new(
  name = "gene_expression",
  assay = "scrna",
  level = "subject",
  format = "rds",
  description = "Gene expression by subject",
  path_template = "{root}/{subject_id}/expr.rds",
  root_key = "scrna_root",
  reader = "readRDS",
  key_cols = c("subject_id", "gene")
)

# A pair spec for a sample map that labels roles case and control.
spec_cc <- analysis_spec_new(
  name = "case_control",
  assay = "wgs",
  level = "pair",
  path_template = "{root}/{pair_id}.csv",
  root_key = "wgs_root",
  tumor_role = "case",
```

```
  normal_role = "control",
  pair_sep = "_vs_"
)
```

`apply_corrections` *Apply documented corrections to a manifest*

Description

Applies a corrections table to a long-format manifest, one row per sample, and records what changed. A study keeps its overrides in one table with a reason for each, instead of inline edits spread over scripts.

Usage

```
apply_corrections(manifest, corrections)
```

Arguments

`manifest`	A data frame with one row per sample and at least a `subject_id` and a `sample_id` column.
`corrections`	A data frame with columns `level`, `id`, `column`, `value`, and `reason`. `level` is `"subject"` or `"sample"`. `id` names the subject or sample. `column` names the manifest column to change and `value` is the new value. See `read_corrections()` to read one from a file.

Details

A subject correction changes every manifest row for that subject. A sample correction changes the row or rows for that sample. Corrections are applied in order, so a later row can overwrite an earlier one for the same cell. A corrected column becomes character, and an `NA` value clears the cell.

The audit table has one row per correction: `level`, `id`, `column`, `old_value`, `new_value`, `reason`, and `n_rows`, the number of manifest rows that changed. When the old values differ across those rows they are joined with `"; "`. Applying corrections to an already corrected manifest appends to the existing audit table.

The function errors when a required column of `corrections` is missing, when a level is not `"subject"` or `"sample"`, when an id is not in the manifest, or when a column is not in the manifest.

Value

The corrected manifest as a tibble. The `"corrections"` attribute holds the audit table, which `corrections_log()` returns.

See Also

`read_corrections()`, `corrections_log()`

Examples

```
manifest <- tibble::tibble(
  subject_id = c("R1", "R1", "R2"),
  assay = c("wes", "wes", "wes"),
  sample_id = c("R1_T", "R1_N", "R2_T"),
  genotype = c("WT", "WT", "KO"),
  fastq = c("r1_t.fq.gz", "r1_n.fq.gz", "r2_t.fq.gz")
)
corrections <- tibble::tibble(
  level = c("subject", "sample"),
  id = c("R1", "R2_T"),
  column = c("genotype", "fastq"),
  value = c("KO", "r2_tumor.fq.gz"),
  reason = c("genotyping rerun on 2026-03-01", "vendor renamed the file")
)

corrected <- apply_corrections(manifest, corrections)
corrected
corrections_log(corrected)
```

---

as_coldata *Build sample metadata for a count matrix*

Description

Joins a cohort's sample map and subject table for one assay, and returns the result as a base data.frame with row names set to a sample id column. This is the shape `colData` (SummarizedExperiment, DESeq2) and similar analysis objects expect.

Usage

```
as_coldata(cohort, assay, samples = NULL, rownames = "sample_id", ref = NULL)
```

Arguments

`cohort`	A Cohort object.
`assay`	Character scalar naming the assay to include.
`samples`	Optional character vector of sample ids, in the order they should appear (matching, for example, the column order of a count matrix). Every id must be one of the cohort's samples for `assay`; otherwise this errors and lists the ones it could not find.
`rownames`	Character scalar naming the column to use as row names. Default `"sample_id"`.
`ref`	Optional named list. Each name is a column to convert to a factor, and each value the level to use as the reference (first) level, as in `stats::relevel()`. Use it to set a control or wild-type group as the baseline before a differential analysis.

Value

A data.frame with one row per sample, row names set to `rownames`, ordered to match `samples` when given.

See Also

`samples()`, `join_metadata()`

Examples

```
data(example_cohort)
coldata <- as_coldata(example_cohort, assay = "wes")
coldata

as_coldata(example_cohort, assay = "wes", ref = list(genotype = "WT"))
```

---

check_paths *Check that a cohort's file paths exist on disk*

Description

Tests every path a cohort references: the file-like columns of its sample map (`fastq_1`, `bam`, and similar), and every entry of `cohort@paths`. Never errors; a summary is reported when something is missing.

Usage

```
check_paths(cohort, cols = NULL)
```

Arguments

`cohort`	A Cohort object.
`cols`	Optional character vector of `sample_map` column names to check as paths, overriding the default set of recognized path columns (`fastq_1`, `fastq_2`, `bam`, `cram`, `vcf`, `matrix_dir`, `h5`).

Value

A tibble with one row per checked path and columns `source` (`"sample_map"` or `"paths"`), `key` (`sample_id`, or the name in `cohort@paths`), `column` (the `sample_map` column, `NA` for a `cohort@paths` entry), `path`, and `exists`. `exists` is `NA` for a missing (`NA`) path, so an unset path is not read as a broken one.

See Also

`sample_sheet()`, `samples()`

Examples

```
data(example_cohort)
check_paths(example_cohort)
```

Cohort *S7 Cohort class*

Description

An S7 class that keeps the subjects and samples of a study in one object. A Cohort holds a subject table, a long-format sample map, an optional Study, file paths, analysis tables, and a registry of analysis specs.

Usage

```
Cohort(
  study = NULL,
  subject_tbl = tibble::tibble(subject_id = character(), species = character()),
 sample_map = tibble::tibble(subject_id = character(), assay = character(), sample_id =
    character(), role = character()),
  paths = list(),
  analyses = list(),
  registry = list(),
  cache = list(),
 qc = tibble::tibble(scope = character(), id = character(), action = character(), reason
  = character(), previous_status = character(), timestamp = as.POSIXct(character())),
 derived = tibble::tibble(name = character(), from = character(), level = character(),
    cutoffs = list(), n_derived = integer(), n_na = integer(), timestamp =
    as.POSIXct(character()))
)
```

Arguments

study	A Study object with project-level context, or NULL.
subject_tbl	A data frame with one row per subject. Required columns: subject_id and species, both character. Common optional columns: sex, strain, genotype, cohort, timepoint, notes. Checked by validate_cohort(). Defaults to an empty table with the two required columns.
sample_map	A long-format data frame with one row per sample. Required columns: subject_id, assay, sample_id, role, all character. A new assay is a new row, never a new column. Checked by validate_cohort(). Defaults to an empty table with the four required columns.
paths	Named list of file paths to data files or result folders. Defaults to an empty list.
analyses	Named list of analysis tables or other data objects. Defaults to an empty list.

registry	Named list of AnalysisSpec objects. Names match spec@name. Defaults to an empty list.
cache	Named list used to memoize loaded analysis data. Cleared by cohort_filter() on every structural change, since it holds state that can always be recomputed. Defaults to an empty list.
qc	A tibble recording every cohort_qc() call (columns scope, id, action, reason, previous_status, timestamp). Unlike cache, this is a durable record and is not cleared by cohort_filter(). Defaults to an empty table.
derived	A tibble recording every cohort_derive() call (columns name, from, level, cutoffs, n_derived, n_na, timestamp). Durable in the same way as qc. Defaults to an empty table.

Details

Use cohort_new() to build a Cohort. It checks the input types, converts both tables to tibbles, and runs validate_cohort(). Construction itself also checks subject_tbl and sample_map with the same rules, so building a Cohort any other way still enforces the required columns.

Subjects live only in subject_tbl. Use subject() to read one row as a Subject object.

Access properties with the @ operator:

```
cohort@study         # Study object or NULL
cohort@subject_tbl   # Subject metadata table
cohort@sample_map    # Sample mapping table
cohort@paths         # File paths
cohort@analyses      # Stored analysis results
cohort@registry      # Named list of AnalysisSpec objects
cohort@cache         # Memoization cache
cohort@qc            # QC audit log
cohort@derived       # Derived-column provenance
```

Value

A Cohort object with the given properties.

See Also

cohort_new() for object construction, subject() for reading one subject, validate_cohort() for validation details, validate_manifest() for manifest preparation, read_manifest_csv() for loading manifest from file, analysis_register() for registering analyses

Examples

```
# An empty cohort has the required columns and nothing else.
empty <- Cohort()
empty@subject_tbl
empty@sample_map

# The raw constructor runs the same checks as cohort_new().
```

```
cohort <- Cohort(
  subject_tbl = data.frame(subject_id = "R1", species = "rat"),
  sample_map = data.frame(
    subject_id = "R1", assay = "wes", sample_id = "T1", role = "tumor"
  )
)
cohort
```

`cohort_contrasts` *Every pairwise contrast between a cohort's groups*

Description

Enumerates every pairwise combination of the groups in `x`, so a caller can turn each pair into a comparison (for example, filtering a cohort down to one side and diffing an analysis table against the other).

Usage

```
cohort_contrasts(x, by = NULL)
```

Arguments

`x`	Either a Cohort (then `by` is required, and `cohort_groups()` runs internally) or the tibble `cohort_groups()` already returned (then `by` must not be given).
`by`	Character vector of `subject_tbl` columns to group by. Only used when `x` is a Cohort.

Details

This is deliberately minimal: it does not repeat the raw `by` values on each row. Join back to `cohort_groups()`'s output on `group_label` for those.

Value

A tibble with one row per pair: `group_a`, `group_b` (the two groups' labels), `subject_ids_a`, `subject_ids_b` (list-columns of subject ids), and `n_a`, `n_b` (their sizes).

See Also

`cohort_groups()`, `cohort_filter()`

Examples

```
data(example_cohort)
contrasts <- cohort_contrasts(example_cohort, by = "species")
contrasts

cohort_filter(example_cohort, subject_ids = contrasts$subject_ids_a[[1]])
```

`cohort_derive` *Derive a column from cutoffs on an existing numeric column*

Description

Bins an existing numeric column into named groups at one or more cutoff values, and writes the result as a new column, so a rule like "early onset is 120 days or under" is a value you pass in, not code you write.

Usage

```
cohort_derive(cohort, name, from, cutoffs, level)
```

Arguments

`cohort`	A Cohort object.
`name`	Character scalar naming the new column.
`from`	Character scalar naming an existing numeric (or numeric-coercible) column to bin.
`cutoffs`	A named numeric vector. Sorted internally, so the order given does not matter. See Details for how names become bins.
`level`	One of `"subject"` or `"sample"`: whether `from` and `name` act on `subject_tbl` or `sample_map`. No default.

Details

Sort `cutoffs` by value. Bin `i` is everything up to and including `cutoffs[i]`, labeled `names(cutoffs)[i]`. The bin above the highest cutoff is `NA`, unless the highest cutoff is itself `Inf` with its own name, in which case that bin is fully covered.

So `cutoffs = c(early_onset = 120)` gives two groups: `"early_onset"` for values at or under 120, and `NA` above it, a legitimate pattern when only the named group matters. To cover every value, name the top bin too: `cutoffs = c(early_onset = 120, late_onset = Inf)`. More than one real cutoff works the same way, at no extra cost.

A value in `from` that is not already missing but fails to parse as numeric is an error naming the offending ids, never a silent `NA`. A value that was already missing stays `NA` in the derived column.

Re-deriving with the same `name` overwrites the column, the same way `analysis_register()` replaces a spec of the same name. Every call is recorded in `derive_log()`, which, like `qc_log()`, is not cleared by `cohort_filter()`.

Value

A new Cohort.

See Also

derive_log(), cohort_groups()

Examples

```
manifest <- data.frame(
  subject_id = c("S1", "S2", "S3"),
  species = "rat",
  onset_days = c(90, 150, 200),
  assay = "wes", sample_id = c("a", "b", "c"), role = "tumor"
)
parsed <- validate_manifest(manifest)
cohort <- cohort_new(parsed$subject_tbl, parsed$sample_map)

# Only the named group matters; everything above 120 is NA.
only_named <- cohort_derive(
  cohort, "early_only",
  from = "onset_days", cutoffs = c(early_onset = 120), level = "subject"
)
subjects(only_named)

# Fully covering two groups with an Inf-capped top cutoff.
covered <- cohort_derive(
  cohort, "onset_group",
  from = "onset_days", cutoffs = c(early = 120, late = Inf),
  level = "subject"
)
subjects(covered)
derive_log(covered)
```

---

cohort_filter *Keep a subset of a cohort's subjects or assays*

Description

Filters a cohort's subject table and sample map together, so the result stays a valid Cohort. Any loaded analysis table that has a `subject_id` column is filtered to match; the registry and paths are kept as they are.

Usage

```
cohort_filter(
  cohort,
  ...,
```

```
  subject_ids = NULL,
  assays = NULL,
  drop_sample_ids = NULL,
  drop_empty = TRUE
)
```

Arguments

cohort	A Cohort object.
...	Data-masked filter expressions evaluated against cohort@subject_tbl, as in dplyr::filter(). Optional.
subject_ids	Optional character vector. Keep only these subject ids.
assays	Optional character vector. Keep only sample rows with these assays.
drop_sample_ids	Optional character vector. Remove sample rows with these sample ids. Unlike subject_ids and assays, which both keep a match, this one drops a match: it is the only way to remove specific samples without also naming every sample to keep.
drop_empty	Logical. When TRUE (default), a subject left with no sample after the assays/drop_sample_ids filters is also removed from subject_tbl. When FALSE, such a subject is kept with no rows in sample_map.

Details

The five ways to narrow a cohort combine: ... and subject_ids both narrow subject_tbl, and assays/drop_sample_ids narrow sample_map. sample_map is always restricted to the subjects that remain in subject_tbl after ... and subject_ids, regardless of drop_empty.

Value

A new Cohort. The cache is reset, since it can hold loaded data or a translation result computed for the full set of subjects.

See Also

subjects(), samples(), cohort_new()

Examples

```
data(example_cohort)

# By an expression on subject_tbl
cohort_filter(example_cohort, species == "rat")

# By explicit ids
cohort_filter(example_cohort, subject_ids = c("RAT001", "MOUSE001"))

# By assay, dropping subjects left with no sample
cohort_filter(example_cohort, assays = "scrna")
```

```
# By excluding specific sample ids (e.g. samples that failed QC)
bad_id <- samples(example_cohort)$sample_id[[1]]
cohort_filter(example_cohort, drop_sample_ids = bad_id)
```

---

cohort_groups *Group a cohort's subjects by one or more columns*

Description

Groups `cohort@subject_tbl` by the given columns and returns one row per combination that actually occurs, with the matching subject ids.

Usage

```
cohort_groups(cohort, by)
```

Arguments

`cohort`	A Cohort object.
`by`	Character vector of one or more `subject_tbl` column names to group by. Required, no default.

Details

Only combinations that occur in `subject_tbl` appear; this never invents a row for a combination of values that no subject has. An `NA` in a `by` column forms its own group rather than being dropped.

Value

A tibble with the `by` columns, `group_label` (the `by` values pasted together with `"/"`), `n` (the number of subjects), and `subject_ids` (a list-column of sorted, unique subject ids).

See Also

`cohort_contrasts()`, `subjects()`

Examples

```
data(example_cohort)
cohort_groups(example_cohort, by = "species")
```

cohort_new *Create a Cohort object*

Description

Builds a Cohort from a subject table and a sample map, with an optional Study, file paths, and analysis tables. The two tables are usually the output of `validate_manifest()`.

Usage

```
cohort_new(
  subject_tbl,
  sample_map,
  study = NULL,
  paths = list(),
  analyses = list()
)
```

Arguments

`subject_tbl`	A data frame with one row per subject. Required columns: `subject_id` and `species`, both character. Other columns are kept as given.
`sample_map`	A long-format data frame with one row per sample. Required columns: `subject_id`, `assay`, `sample_id`, `role`, all character.
`study`	A Study object, or NULL. Defaults to NULL.
`paths`	Named list of file paths to data files or result folders. Defaults to an empty list.
`analyses`	Named list of analysis tables or other data objects. Defaults to an empty list.

Details

The steps are:

1. Check that `subject_tbl` and `sample_map` are data frames.
2. Convert both to tibbles.
3. Build the Cohort.
4. Run `validate_cohort()`.

The function does not build Subject objects. Use `subject()` to read one subject from the cohort when an object is needed.

Value

A Cohort object. An error when the tables fail `validate_cohort()`.

See Also

validate_manifest() for preparing input tables, validate_cohort() for the checks, subject() for reading one subject, Study for study metadata

Examples

```
study <- study_new(
  study_id = "STUDY001",
  title = "Cross-species study",
  assays = c("WES", "snRNA-seq")
)

# A long-format manifest: one row per sample
manifest <- data.frame(
  subject_id = c("RAT001", "RAT001", "MOUSE1", "MOUSE1"),
  species = c("rat", "rat", "mouse", "mouse"),
  sex = c("M", "M", "F", "F"),
  assay = c("wes", "scrna", "wes", "atac"),
  sample_id = c("WES_T1", "RNA_1", "WES_T2", "ATAC_1"),
  role = c("tumor", "tumor", "tumor", NA)
)

parsed <- validate_manifest(manifest)
cohort <- cohort_new(
  study = study,
  subject_tbl = parsed$subject_tbl,
  sample_map = parsed$sample_map
)
print(cohort)

# Read one subject as a Subject object
subject(cohort, "RAT001")
```

---

cohort_qc *Flag or drop subjects or samples for quality control*

Description

Records a QC decision against a cohort: either annotate matching rows with a status and reason (action = "flag"), or remove them (action = "drop"). Every call is recorded in the cohort's QC log (see qc_log()), which is not cleared by cohort_filter(), so the record of why something was dropped survives later structural changes.

Usage

```
cohort_qc(cohort, ids, scope, action, reason)
```

Arguments

cohort	A Cohort object.
ids	Character vector of ids to act on. At least one, no NA. Duplicates are removed. Their meaning depends on scope.
scope	One of "sample" or "subject": whether ids are sample ids (matched against cohort@sample_map$sample_id) or subject ids (matched against cohort@subject_tbl$subject_id). No default.
action	One of "flag" or "drop". "flag" sets qc_status/ qc_reason on the matching rows and keeps them. "drop" removes the matching rows entirely. No default.
reason	A single, non-empty string explaining the decision.

Details

An id in ids that does not exist for the given scope is an error; it is never silently ignored.

action = "flag", scope = "sample" sets qc_status/qc_reason on sample_map, creating the columns if they are absent. scope = "subject" sets the same two column names on subject_tbl instead; these are a separate pair of columns from the sample-level ones, and both can be set on the same cohort for different reasons. Flagging an id that already has a qc_reason appends the new reason rather than replacing it.

action = "drop" delegates to cohort_filter(): scope = "sample" uses its drop_sample_ids argument, with drop_empty = FALSE so a subject left with no samples is not also removed; scope = "subject" uses its subject_ids argument to keep every other subject. Either way, the resulting cache reset is the same one a direct cohort_filter() call would produce.

sample_id is normally unique, so qc_log()'s previous_status reflects the one matching row. If sample_map holds duplicate sample_ids (built with allow_duplicates = TRUE), every matching row is still flagged or dropped correctly, but the logged previous_status reflects only one of them.

Value

A new Cohort.

See Also

qc_log(), cohort_filter()

Examples

```
data(example_cohort)

# Flag one sample, keeping it
bad_id <- samples(example_cohort)$sample_id[[1]]
flagged <- cohort_qc(
  example_cohort, bad_id,
  scope = "sample", action = "flag", reason = "failed QC review"
)
samples(flagged)
qc_log(flagged)
```

```
# Drop the same sample instead
dropped <- cohort_qc(
  example_cohort, bad_id,
  scope = "sample", action = "drop", reason = "failed QC review"
)
nrow(samples(dropped)) < nrow(samples(example_cohort))
```

`cohort_read` *Read a cohort saved with cohort_save()*

Description

Reads the RDS file, checks it is a cohort file, and re-validates the cohort before returning it.

Usage

```
cohort_read(path)
```

Arguments

`path`	Path to a file written by `cohort_save()`.

Value

The Cohort object.

See Also

`cohort_save()`

Examples

```
data(example_cohort)
path <- tempfile(fileext = ".rds")
cohort_save(example_cohort, path)

restored <- cohort_read(path)
identical(subjects(restored), subjects(example_cohort))
unlink(path)
```

cohort_save *Save a cohort to an RDS file*

Description

Wraps a Cohort with a small format marker and the package version, and writes it with `saveRDS()`. Read it back with `cohort_read()`.

Usage

```
cohort_save(cohort, path)
```

Arguments

`cohort`	A Cohort object.
`path`	Output file path, conventionally ending in `.rds`.

Value

`path`, invisibly.

See Also

`cohort_read()`, `write_manifest()`

Examples

```
data(example_cohort)
path <- tempfile(fileext = ".rds")
cohort_save(example_cohort, path)
cohort_read(path)
unlink(path)
```

completeness *Per-assay sample counts for a cohort*

Description

Summarizes how many samples each subject has for each assay. This is the table most studies ask for first: which subjects are missing which assay.

Usage

```
completeness(cohort, wide = FALSE)
```

Arguments

cohort	A Cohort object.
wide	Logical. When FALSE (default), returns one row per subject_id x assay with an n_samples count. When TRUE, pivots to one row per subject and one column per assay, with 0 where a subject has no sample for that assay.

Details

The wide form always has one row per subject in cohort@subject_tbl, even a subject with zero samples for every assay, and one column per assay that appears anywhere in cohort@sample_map.

Value

A tibble. See wide above for its shape.

See Also

subjects(), samples(), validate_manifest()

Examples

```
data(example_cohort)
completeness(example_cohort)
completeness(example_cohort, wide = TRUE)
```

---

corrections_log *Return the audit table of a corrected manifest*

Description

Reads the "corrections" attribute that apply_corrections() sets.

Usage

```
corrections_log(x)
```

Arguments

x	A manifest, corrected or not.

Value

A tibble with columns level, id, column, old_value, new_value, reason, and n_rows. It has no rows when x has not been corrected.

Examples

```
manifest <- tibble::tibble(subject_id = "R1", sample_id = "R1_T")
corrections_log(manifest)
```

`derive_log` *Read a cohort's derived-column log*

Description

A thin, named accessor for `cohort@derived`, the provenance every `cohort_derive()` call appends to.

Usage

```
derive_log(cohort)
```

Arguments

`cohort`	A Cohort object.

Value

`cohort@derived` as a tibble.

See Also

`cohort_derive()`

Examples

```
data(example_cohort)
derive_log(example_cohort)
```

`ensure_dir` *Create a folder when it is absent*

Description

Creates `path` and its parents with `fs::dir_create()` when the folder does not exist yet. An existing folder is left alone.

Usage

```
ensure_dir(path)
```

Arguments

path	Folder to create.

Value

path, invisibly.

Examples

```
out <- fs::path(tempfile(), "results", "qc")
ensure_dir(out)
fs::dir_exists(out)

unlink(fs::path_dir(fs::path_dir(out)), recursive = TRUE)
```

---

example_cohort *Example Cohort Dataset*

Description

A small Cohort with two rat and two mouse subjects. Use it to explore the data model, to try the API, or as a template for a cohort built from real data.

Usage

```
example_cohort
```

Format

A Cohort object (S7 class) with the following structure:

- study: A Study object with metadata for a cross-species genomics project
- subject_tbl (tibble): 4 subjects (2 rat, 2 mouse) with species, sex, strain, genotype, cohort, timepoint
- sample_map (tibble): Long-format map (subject_id, assay, sample_id, role, fastq_1, fastq_2) covering WES tumor/normal and snRNA-seq samples. fastq_1/fastq_2 show that an extra sample-level column survives validate_manifest() alongside the four canonical ones.
- paths (list): Empty, ready for file paths
- analyses (list): Empty, ready for analysis results

Details

The cohort shows:

- Two species in one subject table
- Two assays per subject in one long-format sample map
- A Study object for project context

Subjects are stored as rows of subject_tbl. Use subject() to read one of them as a Subject object.

See Also

cohort_new() for creating Cohort objects, subject() for reading one subject, validate_manifest() for preparing manifest data, read_manifest_csv() for loading manifest from CSV file

Examples

```
# Load the example cohort
data(example_cohort)

# View the study metadata
example_cohort@study

# Read one subject as a Subject object
rat1 <- subject(example_cohort, "RAT001")
rat1@species
rat1@sex

# List all subject ids
example_cohort@subject_tbl$subject_id

# View all subjects with metadata
example_cohort@subject_tbl

# View the sample map
example_cohort@sample_map

# Count subjects by species
table(example_cohort@subject_tbl$species)
```

---

join_metadata *Add cohort metadata to an analysis object*

Description

Joins a cohort's sample and subject metadata onto a data.frame, a SummarizedExperiment, or a Seurat object, matched by sample id.

Usage

```
join_metadata(object, cohort, assay = NULL, by = "sample_id", col = NULL)
```

Arguments

object	A data.frame, a SummarizedExperiment, or a Seurat object.
cohort	A Cohort object.

assay	Optional character scalar restricting the join to one assay's samples. Defaults to NULL, which matches by against every sample in the cohort; this is usually enough, since sample_id is unique across the whole cohort unless the manifest allowed duplicates.
by	Character scalar naming the sample id column to join on. Default "sample_id". For a data.frame, this must be a column of object. For a SummarizedExperiment or a Seurat object, by names the column of samples(cohort, with_subjects = TRUE) to match against (see col).
col	For a SummarizedExperiment, an optional column of its colData holding sample ids; defaults to colnames(object). For a Seurat object, an optional column of its meta.data holding sample ids; defaults to "orig.ident".

Details

A SummarizedExperiment or Seurat column that does not match any sample in the cohort is an error, naming the unmatched ids. A data.frame join is a plain left join, so it keeps every row of object and leaves an unmatched row's new columns as NA.

Value

object, with the cohort's metadata columns added: joined columns for a data.frame, added colData columns for a SummarizedExperiment, added meta.data columns for a Seurat object.

See Also

as_coldata(), samples()

Examples

```
data(example_cohort)
expr <- data.frame(
  sample_id = example_cohort@sample_map$sample_id[1:2],
  value = c(1, 2)
)
join_metadata(expr, example_cohort)
```

---

liftover_backends *List registered liftover backends*

Description

List registered liftover backends

Usage

```
liftover_backends()
```

Value

A character vector of registered backend names.

See Also

`register_liftover_backend()`, `liftover_intervals()`

Examples

```
liftover_backends()
```

`liftover_crossmap` *Liftover backend backed by CrossMap*

Description

Liftover backend that shells out to the external CrossMap tool (`CrossMap bed`). CrossMap must be installed and on the `PATH`. For most interval workflows the R-native `liftover_rtracklayer()` backend is sufficient and easier to deploy; CrossMap is most valuable for allele-aware variant translation (see `liftover_vcf()`).

Usage

```
liftover_crossmap(intervals, chain, crossmap = NULL, ...)
```

Arguments

`intervals`	A tibble of intervals (see `liftover_rtracklayer()`).
`chain`	Path to a chain file.
`crossmap`	Path or name of the CrossMap executable. Defaults to auto-detection on the `PATH`.
`...`	Unused.

Value

A list with `mapped` and `unmapped` tibbles.

See Also

`liftover_intervals()`, `liftover_rtracklayer()`, `liftover_vcf()`

Examples

```
if (nzchar(Sys.which("CrossMap")) || nzchar(Sys.which("CrossMap.py"))) {
  chain <- tempfile(fileext = ".chain")
  writeLines(
    c(
      "chain 1000 chr1 100000 + 0 1000 chrT 200000 + 10000 11000 1",
      "1000",
      ""
    ),
    chain
  )
  ints <- tibble::tibble(
    seqnames = c("chr1", "chr1"),
    start = c(100, 5000),
    end = c(200, 5100),
    .feature_id = 1:2
  )
  out <- liftover_crossmap(ints, chain)
  out$mapped
  unlink(chain)
}
```

`liftover_intervals` *Liftover a set of genomic intervals across assemblies or species*

Description

Translates coordinate features (intervals such as variants, peaks, or regions) from one genome assembly or species to another using a chain file, via a pluggable backend. Returns a TranslationResult that keeps both the mapped and the unmapped features, so loss is explicit.

Usage

```
liftover_intervals(
  intervals,
  chain,
  from = NA_character_,
  to = NA_character_,
  backend = "rtracklayer",
  ...
)
```

Arguments

`intervals`	A data.frame or tibble of intervals. Required columns: `seqnames` (character), `start` (integer), `end` (integer). Optional `strand`; any additional columns are preserved on unmapped rows.

chain	Path to a chain file (e.g. a UCSC .chain/.chain.gz), or a backend-specific chain object. Cross-species chains (e.g. rat-to-human) enable cross-species liftover where synteny permits.
from, to	Optional character scalars recording the source and target assembly/species for provenance.
backend	Either the name of a registered backend (see liftover_backends()) or a backend function. Defaults to "rtracklayer".
...	Additional arguments passed to the backend.

Details

Cross-species liftover is inherently lossy and limited to syntenic, alignable regions; non-conserved regions (and many regulatory elements) will not map. Always inspect translation_stats() and the unmapped table rather than assuming full recovery. For allele-aware variant (VCF) translation, see liftover_vcf().

Value

A TranslationResult.

See Also

liftover_rtracklayer(), liftover_crossmap(), translate()

Examples

```
ints <- data.frame(
  seqnames = c("chr1", "chr1"),
  start = c(100, 5000),
  end = c(200, 5100)
)
# Illustrative in-memory backend (maps the first interval, drops the rest):
backend <- function(intervals, chain, ...) {
  list(
    mapped = tibble::tibble(
      .feature_id = intervals$.feature_id[1],
      seqnames = "chrT", start = 1L, end = 100L, strand = "*"
    ),
    unmapped = intervals[-1, , drop = FALSE]
  )
}
res <- liftover_intervals(ints, chain = "none", to = "human", backend = backend)
translation_stats(res)
```

`liftover_rtracklayer` *Liftover backend backed by rtracklayer*

Description

R-native liftover backend using `rtracklayer::liftOver()` with a UCSC chain file. This is the default backend for `liftover_intervals()`; it requires no external tools but needs the Bioconductor packages `rtracklayer`, `GenomicRanges`, `IRanges`, and `S4Vectors`.

Usage

```
liftover_rtracklayer(intervals, chain, ...)
```

Arguments

`intervals`	A tibble of intervals with `seqnames`, `start`, `end`, an optional `strand`, and a `.feature_id` key (supplied by `liftover_intervals()`).
`chain`	Path to a UCSC chain file.
`...`	Unused.

Value

A list with `mapped` and `unmapped` tibbles.

See Also

`liftover_intervals()`, `liftover_crossmap()`

Examples

```
# rtracklayer and the packages it depends on take a few seconds to load.
if (
  requireNamespace("rtracklayer", quietly = TRUE) &&
    requireNamespace("GenomicRanges", quietly = TRUE)
) {
  chain <- tempfile(fileext = ".chain")
  writeLines(
    c(
      "chain 1000 chr1 100000 + 0 1000 chrT 200000 + 10000 11000 1",
      "1000",
      ""
    ),
    chain
  )
  ints <- tibble::tibble(
    seqnames = c("chr1", "chr1"),
    start = c(100, 5000),
    end = c(200, 5100),
```

```
    .feature_id = 1:2
  )
  out <- liftover_rtracklayer(ints, chain)
  out$mapped
  unlink(chain)
}
```

`liftover_vcf` *Liftover a VCF of variants with CrossMap (allele-aware)*

Description

Thin, experimental wrapper around `CrossMap vcf`, which performs allele-aware variant liftover (updating the REF allele against the target genome and handling strand flips) – something plain interval liftover does not do. Requires CrossMap on the `PATH` and a target-genome FASTA.

Usage

```
liftover_vcf(
  vcf,
  chain,
  ref_fasta,
  out = tempfile(fileext = ".vcf"),
  from = NA_character_,
  to = NA_character_,
  crossmap = NULL
)
```

Arguments

`vcf`	Path to the input VCF.
`chain`	Path to a chain file.
`ref_fasta`	Path to the target-genome FASTA.
`out`	Output VCF path. Defaults to a temporary file.
`from`, `to`	Optional provenance strings.
`crossmap`	Path or name of the CrossMap executable. Defaults to auto-detection on the `PATH`.

Details

This function is experimental and intentionally minimal: it runs CrossMap and reports the produced paths and unmapped count rather than parsing variants into R. Parse the output VCF with your tool of choice (e.g. `VariantAnnotation`).

Value

A TranslationResult whose mapped and unmapped carry the output and unmapped VCF paths; stats records the number of unmapped records.

See Also

liftover_intervals(), liftover_crossmap()

Examples

```
## Not run:
# Not run: needs CrossMap on the PATH, a chain file, and the target
# genome as a FASTA file, none of which ship with the package.
res <- liftover_vcf(
  vcf = "calls.rn7.vcf",
  chain = "rn7ToHg38.over.chain",
  ref_fasta = "hg38.fa",
  from = "rn7",
  to = "hg38"
)
res@mapped$path # the lifted VCF
res@unmapped$path # the records CrossMap could not place
res@stats$n_unmapped

## End(Not run)
```

---

load_analyses *Load registered analyses into a cohort from disk*

Description

Loads the feature table for each registered AnalysisSpec (via load_analysis()) and returns a new Cohort with analyses populated. The per-analysis file manifests are stored in the cohort cache and retrievable with analysis_files(). This is the step that takes a cohort from *paths* to *loaded feature tables*, ready for translate().

Usage

```
load_analyses(cohort, analyses = NULL, readers = NULL)
```

Arguments

cohort	A Cohort with registered specs (see analysis_register()).
analyses	Optional character vector restricting which registered analyses to load. Defaults to all that have a path_template.
readers	Optional named list of reader overrides, keyed by analysis name (each a function or "pkg::fun" name).

Details

Specs without a `path_template` are skipped with a warning. Use `analysis_files()` to inspect which files were found or missing.

Value

A new Cohort with `analyses` populated for the loaded specs.

See Also

`load_analysis()`, `analysis_files()`, `translate()`

Examples

```
# One per-subject CSV on disk, one registered spec that points at it.
dir <- tempfile()
dir.create(dir)
write.csv(
  data.frame(gene = "TP53", value = 1), file.path(dir, "S1.csv"),
  row.names = FALSE
)
manifest <- data.frame(
  subject_id = "S1", species = "human", assay = "rna", sample_id = "x"
)
parsed <- validate_manifest(manifest)
cohort <- cohort_new(
  parsed$subject_tbl, parsed$sample_map, paths = list(rna_root = dir)
)
spec <- analysis_spec_new(
  name = "expr", assay = "rna", level = "subject",
  path_template = "{root}/{subject_id}.csv", root_key = "rna_root"
)
cohort <- analysis_register(cohort, spec)

loaded <- load_analyses(cohort)
loaded@analyses$expr
analysis_files(loaded)
unlink(dir, recursive = TRUE)
```

---

load_analysis *Load an analysis's feature table from disk*

Description

Resolves an AnalysisSpec's `path_template` for each unit implied by its `level` (one file per subject, per pair, or one for the whole cohort), reads the existing files with the spec's `reader`, and row-binds them into a single feature table annotated with provenance keys (`subject_id` and/or `pair_id`).

Usage

```
load_analysis(cohort, spec, reader = NULL)
```

Arguments

cohort	A Cohort providing paths (for {root}), subjects, and the sample map (for subject and pair enumeration).
spec	An AnalysisSpec or the name of one registered in cohort.
reader	Optional reader override: a function, or a "fun"/"pkg::fun" name. Defaults to the spec's reader.

Details

Units follow the spec's assay. A subject-level spec enumerates only the subjects with at least one sample of that assay. A pair-level spec calls sample_pairs() with the spec's assay, tumor_role, normal_role, and pair_sep. When no subject or pair matches, the function warns and returns empty tables.

Path tokens supported: {root} (from cohort@paths[[root_key]]), {subject_id}, and for pair-level specs {tumor_sample_id}, {normal_sample_id}, {pair_id} (derived via sample_pairs()). Missing files are skipped (with a warning) and recorded in files, so loading is never silently partial.

The reader comes from the reader argument, else from the spec. The function errors when neither names one. After each file is read, the spec's key_cols must be present in the table (the provenance keys count), or the function errors and names the missing columns.

Value

A list with:

- data: a tibble of all loaded rows (empty if no files were found), with provenance key columns added.
- files: a tibble with one row per unit: its keys, the resolved path, and whether it exists.

See Also

load_analyses(), translate()

Examples

```
# Write a per-subject CSV, then load it.
dir <- tempfile()
dir.create(dir)
write.csv(
  data.frame(gene = "TP53", value = 1), file.path(dir, "S1.csv"),
  row.names = FALSE
)

manifest <- data.frame(
  subject_id = "S1", species = "human", assay = "rna", sample_id = "x"
)
```

```
parsed <- validate_manifest(manifest)
cohort <- cohort_new(
  subject_tbl = parsed$subject_tbl, sample_map = parsed$sample_map,
  paths = list(rna_root = dir)
)

# format, reader, and key_cols come from the template and the level.
spec <- analysis_spec_new(
  name = "expr", assay = "rna", level = "subject",
  path_template = "{root}/{subject_id}.csv", root_key = "rna_root",
  feature_type = "gene"
)
loaded <- load_analysis(cohort, spec)
loaded$data
loaded$files
```

`manifest_from_wide` *Reshape a wide sample table into a long-format manifest*

Description

Many sample sheets start wide: one row per subject, with one column per assay-and-role combination (e.g. `wes_tumor_id`, `wes_normal_id`, `scrna_id`). This turns such a table into the long format `validate_manifest()` expects, one row per non-missing sample id.

Usage

```
manifest_from_wide(x, id_cols, subject_id = "subject_id")
```

Arguments

`x`	A data.frame or tibble, one row per subject.
`id_cols`	A data.frame or tibble describing the columns of `x` that hold sample ids, with columns: • `column`: name of a column in `x`. • `assay`: the assay that column's ids belong to. • `role`: optional; the role of that column's samples. Defaults to `NA` for every row when absent.
`subject_id`	Character scalar naming the subject id column in `x`. Default `"subject_id"`.

Details

A blank or `NA` value in an id column contributes no row, so a subject missing one assay is not stamped with an empty sample id.

Value

A long-format tibble: one row per non-missing sample id in any `id_cols$column`, with `subject_id`, `assay`, `sample_id`, `role`, and every column of `x` that is not in `id_cols$column`. Pass it to `validate_manifest()` next.

See Also

`read_manifest()`, `validate_manifest()`

Examples

```
wide <- data.frame(
  subject_id = c("R1", "R2"),
  species = "rat",
  wes_tumor_id = c("WES_T1", "WES_T2"),
  wes_normal_id = c("WES_N1", NA),
  scrna_id = c("SC_1", "SC_2")
)
id_cols <- data.frame(
  column = c("wes_tumor_id", "wes_normal_id", "scrna_id"),
  assay = c("wes", "wes", "scrna"),
  role = c("tumor", "normal", NA)
)

long <- manifest_from_wide(wide, id_cols)
long
validate_manifest(long)
```

`orthologize` *Deprecated alias for translate()*

Description

`orthologize()` is the earlier name for `translate()`. It still works and calls `translate()` with the same arguments, and warns once per session. New code should call `translate()` directly.

Usage

```
orthologize(x, to, from = NULL, ...)
```

Arguments

`x`	The thing to translate. Either: • a data.frame/tibble of features, or • a Cohort object.
`to`	Character scalar naming the target species/assembly.

from — For a feature table, a character scalar naming the source species/assembly; required for the "ortholog" strategy, optional (recorded as provenance) for "liftover". For a Cohort, NULL (default) infers the source species from subject_tbl$species: used directly when the cohort has one species, or resolved per analysis (and, for an analysis with a subject_id column, per subject) when it has more than one. Give it explicitly to override inference.

... — Strategy-specific arguments. For a **feature table**:

- strategy: "liftover" (coordinate features; see liftover_intervals()) or "ortholog" (gene features; see ortholog_genes()).
- chain: chain-file path for "liftover".
- backend: translation backend (defaults: "rtracklayer" for liftover, "babelgene" for ortholog).
- plus backend arguments such as gene_col/id_type for orthologs.

For a **Cohort**:

- chain: chain-file path used for any feature_type = "interval" analysis. A cohort translated from more than one source species can pass a named list instead, one chain per source species (e.g. list(rat = "rn7ToHg38.chain", mouse = "mm39ToHg38.chain")).
- liftover_backend, ortholog_backend: backends for the two feature kinds.
- analyses: optional character vector restricting which analyses to translate (defaults to all that have a registered spec with a feature_type).

Value

See translate().

See Also

translate()

---

ortholog_babelgene *Ortholog backend backed by babelgene*

Description

Gene-ortholog backend using the offline **babelgene** package, which ships precomputed orthologs between human and a range of model organisms. This is the default backend for ortholog_genes().

Usage

```
ortholog_babelgene(features, from, to, gene_col, id_type, cache = NULL, ...)
```

Arguments

features	A tibble of features with a gene column and a .feature_id key (supplied by ortholog_genes()).
from, to	Source and target species (e.g. "human", "mouse", "rat"). babelgene is human-centric, so model-to-model mappings (e.g. rat-to-mouse) are routed through human.
gene_col	Name of the gene-identifier column.
id_type	One of "symbol", "entrez", "ensembl".
cache	Optional path to a TSV file caching prior lookups. When given, a gene already in the file is read from there instead of queried again, and a newly queried gene is appended for next time. The cache is shared across from/to/id_type combinations in one file.
...	Passed to babelgene::orthologs() (e.g. min_support, top).

Value

A list with mapped and unmapped tibbles.

See Also

ortholog_genes()

Examples

```
if (requireNamespace("babelgene", quietly = TRUE)) {
  feats <- tibble::tibble(gene = c("TP53", "MYC"), .feature_id = 1:2)
  out <- ortholog_babelgene(
    feats,
    from = "human",
    to = "mouse",
    gene_col = "gene",
    id_type = "symbol"
  )
  out$mapped
}
```

---

ortholog_backends *List registered ortholog backends*

Description

List registered ortholog backends

Usage

```
ortholog_backends()
```

Value

A character vector of registered backend names.

See Also

`register_ortholog_backend()`, `ortholog_genes()`

Examples

```
ortholog_backends()
```

`ortholog_genes` *Map gene-level features to orthologs in another species*

Description

Translates gene-level features (a table with a gene-identifier column) from one species to another via a pluggable ortholog backend. Returns a TranslationResult retaining both mapped and unmapped rows, so loss is explicit.

Usage

```
ortholog_genes(
  features,
  from,
  to,
  gene_col = "gene",
  id_type = c("symbol", "entrez", "ensembl"),
  backend = "babelgene",
  ...
)
```

Arguments

`features`	A data.frame or tibble with one column of gene identifiers (`gene_col`). Other columns (e.g. expression values) are preserved on the mapped rows alongside the new `ortholog` column.
`from, to`	Character scalars naming the source and target species (e.g. `"human"`, `"mouse"`, `"rat"`). Both required.
`gene_col`	Name of the column in `features` holding gene identifiers. Defaults to `"gene"`.
`id_type`	Identifier type: one of `"symbol"`, `"entrez"`, `"ensembl"`.
`backend`	Either the name of a registered backend (see `ortholog_backends()`) or a backend function. Defaults to `"babelgene"`.
`...`	Additional arguments passed to the backend.

Details

Ortholog mapping is many-to-many in general: a gene may have zero, one, or several orthologs. Inspect `translation_stats()` and the unmapped table rather than assuming one-to-one correspondence.

Value

A TranslationResult. `mapped` contains the input columns plus an `ortholog` column with target-species identifiers (one input gene may yield multiple ortholog rows).

See Also

`translate()`, `ortholog_babelgene()`, TranslationResult

Examples

```
features <- data.frame(
  gene = c("TP53", "MYC", "NOT_A_GENE"),
  expr = c(1.2, 3.4, 5.6)
)
# Illustrative in-memory backend (uppercases to a fake "ortholog"):
backend <- function(features, from, to, gene_col, id_type, ...) {
  ok <- features[[gene_col]] != "NOT_A_GENE"
  mapped <- features[ok, , drop = FALSE]
  mapped$ortholog <- tolower(mapped[[gene_col]])
  list(mapped = mapped, unmapped = features[!ok, , drop = FALSE])
}
ortholog_genes(features, from = "human", to = "mouse", backend = backend)
```

`project_path` *Build a path under the project root*

Description

Joins path pieces to the project root with `fs::path()`. Nothing is created on disk.

Usage

```
project_path(..., root = project_root())
```

Arguments

`...`	Path pieces, as in `fs::path()`.
`root`	Folder to join to. Defaults to `project_root()`, which is only searched for when `root` is not given.

Value

An `fs_path`.

See Also

`project_root()`, `ensure_dir()`

Examples

```
project_path("data", "manifest.csv", root = "/study")
```

`project_root` *Find the project root folder*

Description

Walks up from `start` until a folder holds one of the `markers`. The default markers are a `.git` entry, a `DESCRIPTION` file, or any file that ends in `.Rproj`. Scripts can then build paths from the root instead of from the working directory.

Usage

```
project_root(start = ".", markers = c(".git", "DESCRIPTION", ".Rproj"))
```

Arguments

`start`	Folder to start from. Defaults to the working directory.
`markers`	Character vector of names to look for. A marker matches an entry with the same name. A marker that starts with a dot, such as `.Rproj`, also matches any entry that ends with it.

Value

The absolute, normalized path of the first folder that holds a marker, as an `fs_path`.

See Also

`project_path()` to join paths under the root.

Examples

```
root <- fs::path(tempfile(), "study")
fs::dir_create(fs::path(root, "scripts", "qc"))
fs::file_create(fs::path(root, "DESCRIPTION"))

project_root(fs::path(root, "scripts", "qc"))

unlink(fs::path_dir(root), recursive = TRUE)
```

qc_log *Read a cohort's QC log*

Description

A thin, named accessor for `cohort@qc`, the audit trail every `cohort_qc()` call appends to.

Usage

```
qc_log(cohort)
```

Arguments

`cohort`	A Cohort object.

Value

`cohort@qc` as a tibble.

See Also

`cohort_qc()`

Examples

```
data(example_cohort)
qc_log(example_cohort)
```

read_corrections *Read a corrections table from a file*

Description

Reads a CSV or TSV file, chosen by extension, with every column as character, and checks that the columns `apply_corrections()` needs are present. An empty value or `NA` in the file becomes `NA`.

Usage

```
read_corrections(path)
```

Arguments

`path`	Path to a `.csv` or `.tsv` file.

Value

A tibble with character columns `level`, `id`, `column`, `value`, and `reason`, plus any other column in the file.

See Also

`apply_corrections()`

Examples

```
path <- tempfile(fileext = ".csv")
writeLines(
  c(
    "level,id,column,value,reason",
    "subject,R1,genotype,KO,genotyping rerun",
    "sample,R2_T,fastq,r2_tumor.fq.gz,vendor renamed the file"
  ),
  path
)

read_corrections(path)

unlink(path)
```

`read_dotenv` *Read a dotenv file into the environment*

Description

Reads `KEY=value` lines from a dotenv file and sets them with `Sys.setenv()`. Blank lines and lines that start with # are skipped. Whitespace around keys and values is trimmed. One pair of matching single or double quotes around a value is removed. A key that is already set in the environment is left alone unless `overwrite = TRUE`. When a key repeats in the file, the last line wins. Values are never printed.

Usage

```
read_dotenv(path = NULL, overwrite = FALSE)
```

Arguments

`path`	Path to the file. Defaults to `.env` in `project_root()`.
`overwrite`	Replace variables that are already set? Defaults to `FALSE`.

Value

A named character vector of every value read from the file, invisibly. Keys that were skipped because they were already set are included.

Examples

```
env_file <- tempfile(fileext = ".env")
writeLines(
  c(
    "# analysis settings",
    "BIOCOHORT_EXAMPLE_THREADS = 4",
    "BIOCOHORT_EXAMPLE_LABEL = 'batch one'"
  ),
  env_file
)

vars <- read_dotenv(env_file)
names(vars)
Sys.getenv("BIOCOHORT_EXAMPLE_LABEL")

Sys.unsetenv(names(vars))
unlink(env_file)
```

`read_manifest` *Read and validate a long-format manifest file*

Description

Reads a manifest from CSV, TSV, or Excel and delegates to `validate_manifest()` for validation and structuring. Every column is read as character, so an id like `"007"` or `"1.10"` is never silently turned into a number.

Usage

```
read_manifest(
  path,
  ...,
  delim = NULL,
  sheet = NULL,
  sample_cols = NULL,
  species = NULL,
  allow_duplicates = FALSE
)
```

Arguments

`path`	Character scalar with the file path. The format is chosen from the file extension (`.csv`, `.tsv`/`.tab`, `.xlsx`/`.xls`), or by counting commas and tabs in the first line for any other extension.
`...`	Additional named arguments passed to the underlying reader: `readr::read_delim()` for a delimited text file, or `readxl::read_excel()` for an Excel file.
`delim`	Optional character scalar overriding delimiter detection for a delimited text file. Ignored for Excel files.

`sheet` Optional sheet name or number, passed to `readxl::read_excel()`. Ignored for a delimited text file.

`sample_cols`, `species`, `allow_duplicates` Passed to `validate_manifest()`.

Details

The file must be in long format with one row per sample. See `validate_manifest()` for the required columns and the full validation rules. Reading an Excel file needs the **readxl** package.

Value

The list returned by `validate_manifest()`: `subject_tbl`, `sample_map`, and `completeness_tbl`.

See Also

`validate_manifest()` for the validation rules, `manifest_from_wide()` for reshaping a wide table first, `cohort_new()` for creating a Cohort from manifest data

Examples

```
manifest_file <- tempfile(fileext = ".csv")
writeLines(
  c(
    "subject_id,species,assay,sample_id,role",
    "RAT001,rat,wes,WES_T1,tumor",
    "RAT001,rat,wes,WES_N1,normal",
    "MOUSE1,mouse,atac,ATAC_1,NA"
  ),
  manifest_file
)

parsed <- read_manifest(manifest_file)
parsed$subject_tbl
parsed$sample_map
```

---

read_manifest_csv *Read and validate a long-format manifest CSV file*

Description

Reads a tidy, long-format manifest CSV (one row per sample) and delegates to `validate_manifest()` for validation and structuring.

Usage

```
read_manifest_csv(path, ..., allow_duplicates = FALSE)
```

Arguments

path	Character scalar with file path to a CSV manifest file. Path must exist and the file must be readable.
...	Additional named arguments passed to readr::read_csv(), such as col_types, skip, comment, etc.
allow_duplicates	Logical. If TRUE, a repeated sample_id is permitted. If FALSE (default), it raises an error. Passed through to validate_manifest().

Details

Superseded by read_manifest(), which also reads TSV and Excel files. This function stays for existing code; new code should call read_manifest() instead. Every column is read as character unless ... supplies its own col_types.

Value

The list returned by validate_manifest(): subject_tbl, sample_map, and completeness_tbl.

See Also

read_manifest() for CSV, TSV, and Excel in one function, validate_manifest() for the validation rules, cohort_new() for creating a Cohort from manifest data

Examples

```
# Create a temporary long-format CSV manifest
manifest_file <- tempfile(fileext = ".csv")
writeLines(
  c(
    "subject_id,species,assay,sample_id,role",
    "RAT001,rat,wes,WES_T1,tumor",
    "RAT001,rat,wes,WES_N1,normal",
    "RAT001,rat,scrna,RNA_1,tumor",
    "MOUSE1,mouse,atac,ATAC_1,NA",
    "HUM01,human,wgs,WGS_T1,tumor"
  ),
  manifest_file
)

# Read and validate the manifest
parsed <- read_manifest_csv(manifest_file)
parsed$subject_tbl
parsed$sample_map
parsed$completeness_tbl
```

`read_study_yaml` *Build a cohort from a study YAML file*

Description

Reads a study, its manifest, its file paths, and its registered analyses from one YAML file, and returns a ready-to-use Cohort. This turns the handful of calls a study's setup script usually makes (`study_new()`, `read_manifest()`, `cohort_new()`, one `analysis_spec_new()` per analysis) into one function call and one file to edit.

Usage

```
read_study_yaml(path, strict = TRUE)
```

Arguments

`path`	Path to the study YAML file.
`strict`	Logical. When `TRUE` (default), an unknown top-level key is an error. Set `FALSE` to read a `study:` block out of a larger configuration file that has other keys of its own.

Details

The file has these top-level keys, all optional except `manifest`:

- `study`: fields for `study_new()` (`study_id`, `title`, `description`, `hypotheses`, `aims`, `assays`, `genome_builds`, `tags`).
- `manifest`: path to the manifest file, read with `read_manifest()`. Required.
- `sample_cols`: extra sample-level columns, passed to `validate_manifest()`.
- `species`: fills a `species` column when the manifest has none.
- `paths`: a map of root name to path, stored as `cohort@paths`.
- `corrections`: path to a corrections file, applied to the manifest with `apply_corrections()` before it is validated.
- `analyses`: a list of `analysis_spec_new()` field sets, one per registered analysis.

Every path (`manifest`, an entry of `paths`, `corrections`) is resolved relative to the YAML file's own directory unless it is already absolute.

Value

A Cohort built from the file.

See Also

`write_study_yaml()`, `read_manifest()`, `cohort_new()`

Examples

```
dir <- tempfile()
dir.create(dir)
writeLines(
  c(
    "subject_id,species,assay,sample_id,role",
    "R1,rat,wes,T1,tumor",
    "R1,rat,wes,N1,normal"
  ),
  file.path(dir, "manifest.csv")
)
writeLines(
  c(
    "study:",
    "  study_id: PILOT",
    "  title: Example pilot",
    "manifest: manifest.csv"
  ),
  file.path(dir, "study.yaml")
)

cohort <- read_study_yaml(file.path(dir, "study.yaml"))
cohort
```

`register_liftover_backend` *Register a liftover backend*

Description

Adds a named liftover backend so it can be selected by name in `liftover_intervals()`. A backend is a function that performs coordinate translation for a set of intervals; this pluggable design lets the package default to an R-native engine while allowing external tools (e.g. CrossMap) to be swapped in.

Usage

```
register_liftover_backend(name, fn)
```

Arguments

`name`	Character scalar naming the backend.
`fn`	A function with signature `function(intervals, chain, ...)` that returns a list with two tibbles: • `mapped`: translated features, including a `.feature_id` column linking each output row to its input row in `intervals`. • `unmapped`: the input rows (carrying `.feature_id`) that produced no output.

Details

`intervals` passed to a backend is guaranteed to have columns `seqnames`, `start`, `end`, an optional `strand`, and a `.feature_id` integer key added by `liftover_intervals()`.

Value

Invisibly, the backend name.

See Also

`liftover_backends()`, `liftover_intervals()`, `liftover_rtracklayer()`, `liftover_crossmap()`

Examples

```
# A backend that maps every interval onto itself, then use it by name.
passthrough <- function(intervals, chain, ...) {
  list(
    mapped = tibble::tibble(
      .feature_id = intervals$.feature_id,
      seqnames = intervals$seqnames,
      start = intervals$start,
      end = intervals$end,
      strand = "*"
    ),
    unmapped = intervals[0, , drop = FALSE]
  )
}
register_liftover_backend("passthrough", passthrough)
"passthrough" %in% liftover_backends()

ints <- data.frame(seqnames = "chr1", start = 100, end = 200)
liftover_intervals(ints, chain = "none", to = "human", backend = "passthrough")
```

`register_ortholog_backend` *Register a gene-ortholog backend*

Description

Adds a named ortholog backend so it can be selected by name in `ortholog_genes()`. A backend performs gene-level cross-species mapping; this pluggable design mirrors the liftover backends used for coordinate features.

Usage

```
register_ortholog_backend(name, fn)
```

Arguments

name	Character scalar naming the backend.
fn	A function with signature `function(features, from, to, gene_col, id_type, ...)` returning a list with two tibbles: • `mapped`: input rows that had at least one ortholog, carrying the `.feature_id` key and an `ortholog` column with the target-species id. • `unmapped`: input rows (carrying `.feature_id`) with no ortholog.

Value

Invisibly, the backend name.

See Also

`ortholog_backends()`, `ortholog_genes()`, `ortholog_babelgene()`

Examples

```
# A backend that upper-cases rodent symbols into human ones, used by name.
to_upper <- function(features, from, to, gene_col, id_type, ...) {
  mapped <- features
  mapped$ortholog <- toupper(mapped[[gene_col]])
  list(mapped = mapped, unmapped = features[0, , drop = FALSE])
}
register_ortholog_backend("to_upper", to_upper)
"to_upper" %in% ortholog_backends()

genes <- data.frame(gene = c("Tp53", "Myc"))
ortholog_genes(genes, from = "rat", to = "human", backend = "to_upper")
```

---

samples *Read the sample map of a cohort*

Description

A thin, named accessor for `cohort@sample_map`, with optional filters and an optional join to the subject table.

Usage

```
samples(cohort, assay = NULL, role = NULL, with_subjects = FALSE)
```

Arguments

cohort	A Cohort object.
assay	Optional character vector. Keep only these assays.
role	Optional character vector. Keep only these roles.
with_subjects	Logical. When TRUE, left-joins the subject table on subject_id, so subject-level columns (species, genotype, ...) sit alongside each sample row. Default FALSE.

Value

A tibble with the sample map, filtered and optionally joined.

See Also

subjects(), completeness(), sample_pairs()

Examples

```
data(example_cohort)
samples(example_cohort, assay = "wes")
samples(example_cohort, role = "tumor", with_subjects = TRUE)
```

---

sample_pairs *Derive sample pairs from a sample map*

Description

Builds tumor/normal (or, more generally, paired) sample combinations from a canonical long-format sample_map. Pairing is assay-agnostic: any assay that records two roles (e.g. WGS, WES) can be paired with the same helper. Within each subject_id x assay, every sample with tumor_role is paired with every sample carrying normal_role.

Usage

```
sample_pairs(
  sample_map,
  assays = NULL,
  tumor_role = "tumor",
  normal_role = "normal",
  sep = "__"
)
```

Arguments

sample_map	A long-format sample map (e.g. from validate_manifest()), with columns subject_id, assay, sample_id, and role. A Cohort's sample_map (i.e. cohort@sample_map) can be passed directly.
assays	Optional character vector of assays to restrict pairing to. NULL (default) pairs within every assay present in sample_map.
tumor_role	Character scalar naming the role treated as the tumor (or "case") side of a pair. Default "tumor".
normal_role	Character scalar naming the role treated as the normal (or "control") side of a pair. Default "normal".
sep	Character scalar placed between the two sample ids in pair_id. Default "__". Use the separator your pipeline puts in its file names.

Value

A tibble with one row per derived pair and columns:

- subject_id: subject the pair belongs to.
- assay: assay the pair was derived within.
- tumor_sample_id: sample id of the tumor_role member.
- normal_sample_id: sample id of the normal_role member.
- pair_id: composite id, paste0(tumor_sample_id, sep, normal_sample_id).

Subjects lacking either role within an assay contribute no rows. When a subject has multiple tumor and/or normal samples in an assay, all tumor x normal combinations are enumerated.

See Also

validate_manifest() for producing a sample_map

Examples

```
manifest <- data.frame(
  subject_id = c("S1", "S1", "S1", "S2"),
  species = c("rat", "rat", "rat", "rat"),
  assay = c("wes", "wes", "scrna", "wes"),
  sample_id = c("T1", "N1", "R1", "N2"),
  role = c("tumor", "normal", "tumor", "normal"),
  stringsAsFactors = FALSE
)
parsed <- validate_manifest(manifest)

# S1 has a WES tumor/normal pair; S2 has only a normal, so it is dropped.
sample_pairs(parsed$sample_map)

# Restrict to specific assays
sample_pairs(parsed$sample_map, assays = "wes")
```

```
# Match a pipeline that names files tumor_vs_normal
sample_pairs(parsed$sample_map, sep = "_vs_")
```

`sample_sheet` *Write a pipeline sample sheet from a cohort*

Description

Builds the sample sheet a pipeline expects, from a cohort's sample map and subject table. A few common shapes ship with the package (see `sample_sheet_templates()`); pass a custom mapping or a function for anything else.

Usage

```
sample_sheet(
  cohort,
  template = "nf-core/rnaseq",
  assay = NULL,
  path = NULL,
  ...
)
```

Arguments

`cohort`	A Cohort object.
`template`	One of: • A character scalar naming a built-in template (see `sample_sheet_templates()`). • A named character vector mapping an output column name to a column of `samples(cohort, with_subjects = TRUE)`, e.g. `c(sample = "sample_id", path = "fastq_1")`. • A function `function(joined, ...)` returning a data.frame, where `joined` is `samples(cohort, assay = assay, with_subjects = TRUE)`. Default `"nf-core/rnaseq"`.
`assay`	Character scalar naming the assay to include. Required when the cohort has more than one assay; optional when it has exactly one.
`path`	Optional output file path. When given, the sheet is written there as CSV and returned invisibly.
`...`	Passed to a function `template`. Ignored for a built-in or a named-vector template.

Details

The built-in templates are:

- `"nf-core/rnaseq"`: `sample`, `fastq_1`, `fastq_2`, `strandedness` (`"auto"` when the manifest has no `strandedness` column).
- `"nf-core/rnavar"`: `sample`, `fastq_1`, `fastq_2`.
- `"nf-core/atacseq"`: `sample`, `fastq_1`, `fastq_2`, `replicate` (`1` when the manifest has no `replicate` column).
- `"nf-core/sarek"`: `patient`, `sex` (`"XX"`/`"XY"`, from a `sex` column of `"F"`/`"M"`), `status` (`1` for a `"tumor"` or `"resistant"` role, `0` otherwise), `sample`, `lane` (`1` when absent), `fastq_1`, `fastq_2`.

Every template needs `fastq_1` in the sample map (and `fastq_2` where the template writes it); declare it with `validate_manifest(sample_cols = )` or `read_manifest(sample_cols = )` if your manifest names it differently.

Value

A tibble with one row per sample. Invisible when `path` is given.

See Also

`sample_sheet_templates()`, `samples()`, `check_paths()`

Examples

```
manifest <- data.frame(
  subject_id = c("R1", "R1"),
  species = "rat",
  sex = "F",
  assay = "wes",
  sample_id = c("T1", "N1"),
  role = c("tumor", "normal"),
  fastq_1 = c("t1_R1.fq.gz", "n1_R1.fq.gz"),
  fastq_2 = c("t1_R2.fq.gz", "n1_R2.fq.gz"),
  stringsAsFactors = FALSE
)
parsed <- validate_manifest(manifest)
cohort <- cohort_new(parsed$subject_tbl, parsed$sample_map)

sample_sheet(cohort, template = "nf-core/sarek")

# A custom mapping
sample_sheet(cohort, template = c(sample = "sample_id", read1 = "fastq_1"))
```

sample_sheet_templates *List the built-in sample sheet templates*

Description

List the built-in sample sheet templates

Usage

```
sample_sheet_templates()
```

Value

A character vector of template names accepted by `sample_sheet()`'s `template` argument.

See Also

`sample_sheet()`

Examples

```
sample_sheet_templates()
```

Study *S7 Study class*

Description

An immutable S7 class for storing project-level metadata for a study. Study objects provide high-level context and configuration for cohorts and analyses, for any organism and any omics assay.

Usage

```
Study(
  study_id = character(0),
  title = character(0),
  description = NA_character_,
  hypotheses = character(0),
  aims = character(0),
  assays = character(0),
  genome_builds = list(),
  created_at = Sys.time(),
  tags = character(0)
)
```

Arguments

study_id	Character scalar for study identifier. Unique within a project.
title	Character scalar for study name/title.
description	Character scalar for longer description of study purpose, design, or protocols. Optional.
hypotheses	Character vector of research hypotheses. Accepts multiple hypotheses. Optional.
aims	Character vector of specific research aims. Accepts multiple aims. Optional.
assays	Character vector of assay types used (e.g., "wes", "scrna"). Optional.
genome_builds	Named list mapping species to genome build versions, e.g. list(rat = "rn7", mouse = "mm39", human = "hg38"). Any species name and any build string are accepted. Optional.
created_at	POSIXct timestamp for creation. Defaults to the time the object is built.
tags	Character vector of arbitrary tags for categorization. Optional.

Details

Use study_new() to construct Study objects with immediate validation. Construction also validates study_id and title directly, so building a Study any other way still enforces the two required fields.

Access properties via the @ operator:

```
study@study_id
study@title
study@description
study@hypotheses
study@aims
study@assays
study@genome_builds
study@created_at
study@tags
```

Value

A Study object with the given properties.

See Also

study_new() for object construction, Cohort for combining studies with subject data

Examples

```
# The raw constructor; study_new() is the usual way in.
study <- Study(study_id = "PILOT", title = "Pilot study")
study@study_id
study@assays # empty until set
```

study_new *Create a Study object*

Description

Constructs a Study object to describe the overall research project, including study metadata, research hypotheses and aims, assay types, and genome build information. Studies serve as the container for cohorts and provide context for cross-species genomics analysis.

Usage

```
study_new(
  study_id,
  title,
  description = NA_character_,
  description_file = NULL,
  hypotheses = character(),
  aims = character(),
  assays = character(),
  genome_builds = list(),
  created_at = Sys.time(),
  tags = character()
)
```

Arguments

study_id	Character scalar providing a unique identifier for the study. Must be at least 1 character long.
title	Character scalar with the study name/title. Must be at least 1 character long.
description	Character scalar with an optional longer description of the study purpose and design. Always read as plain text. Defaults to NA. Use description_file to read the text from a file instead.
description_file	Optional path to a text file whose content becomes description. When given, description is ignored. Defaults to NULL.
hypotheses	Character vector of research hypotheses. Accepts multiple hypotheses. Optional and defaults to empty vector.
aims	Character vector of specific research aims. Accepts multiple aims. Optional and defaults to empty vector.
assays	Character vector of assay types used in the study (e.g., "WES", "snRNA-seq"). Optional and defaults to empty vector.
genome_builds	Named list mapping species names to genome build versions (e.g., list(rat = "rn7", mouse = "mm10")). Supports rn6, rn7 for rat; mm9, mm10, mm39 for mouse; hg19, hg38 for human. Optional and defaults to empty list.
created_at	POSIXct timestamp for study creation. Defaults to current time.

`tags` Character vector of arbitrary tags for categorization. Optional and defaults to empty vector.

Details

Study objects are S7 classes that immutably store research project metadata. They provide context for cohorts and support cross-species genomics analysis. The study_id and title are required; all other fields are optional.

`description` is always plain text, never a path. To store the content of a README or protocol file, pass its path as `description_file`; the file is read and its content becomes `description`. An error names the path when the file does not exist.

Value

A Study object containing the provided metadata.

See Also

Cohort for combining studies with subject data

Examples

```
# Example with multiple hypotheses and aims
study <- study_new(
  study_id = "STUDY001",
  title = "Cross-species genomics comparison",
  description = "Comparing rat and mouse genomes",
  hypotheses = c(
    "Orthologous genes show conserved expression patterns",
    "Disease genes are enriched in specific pathways"
  ),
  aims = c(
    "Map regulatory regions across species",
    "Identify conserved non-coding elements"
  ),
  assays = c("WES", "snRNA-seq"),
  genome_builds = list(rat = "rn7", mouse = "mm10", human = "hg38")
)
print(study)

# Example with a file as the description
readme <- tempfile(fileext = ".md")
writeLines("# My Study\n\nBackground and design.", readme)
study2 <- study_new(
  study_id = "STUDY002",
  title = "My Study",
  description_file = readme
)
study2@description
```

`Subject` *S7 Subject class*

Description

An immutable S7 class for storing individual-level metadata in cross-species genomics studies. Subject objects represent individual animals or biological samples and are grouped into Cohort objects for collective analysis.

Usage

```
Subject(
  subject_id = character(0),
  species = character(0),
  sex = NA_character_,
  strain = NA_character_,
  genotype = NA_character_,
  cohort = NA_character_,
  timepoint = NA_character_,
  notes = NA_character_
)
```

Arguments

`subject_id`	Character scalar for unique subject identifier.
`species`	Character scalar naming the species. Any value is allowed; `subject_new()` stores it lower-cased.
`sex`	Character scalar for biological sex (e.g., "M", "F"). Optional.
`strain`	Character scalar for strain or breed designation. Optional.
`genotype`	Character scalar for genetic background or modification (e.g., "WT", "KO"). Optional.
`cohort`	Character scalar for cohort membership or treatment group. Optional.
`timepoint`	Character scalar for study timepoint or collection date. Optional.
`notes`	Character scalar for free-form annotations. Optional.

Details

Use `subject_new()` to construct Subject objects; it lower-cases `species`. Construction also validates that `subject_id` and `species` are present, so building a `Subject` any other way still enforces the two required fields. Individual subjects are typically read from a Cohort with `subject()`.

Access properties via the `@` operator:

```
subject@subject_id
subject@species
```

```
subject@sex
subject@strain
subject@genotype
subject@cohort
subject@timepoint
subject@notes
```

Value

A `Subject` object with the given properties.

See Also

`subject_new()` for object construction, Cohort for managing groups of subjects

Examples

```
# The raw constructor; subject_new() is the usual way in.
rat <- Subject(subject_id = "R1", species = "rat", sex = "F")
rat@species
rat@genotype # NA until set
```

`subject` *Build one Subject from a cohort*

Description

Reads the row of `cohort@subject_tbl` whose `subject_id` equals `id` and returns it as a Subject object. A cohort stores its subjects as a table. Use this function when one subject is needed as an object.

Usage

```
subject(cohort, id)
```

Arguments

`cohort`	A Cohort object.
`id`	Character scalar with the `subject_id` to look up.

Details

The Subject fields are read from the columns `subject_id`, `species`, `sex`, `strain`, `genotype`, `cohort`, `timepoint`, and `notes`. Values are coerced to character and missing values stay missing. A column that is absent from `subject_tbl` gives `NA`. Other columns are ignored.

An unknown `id` is an error. The error lists up to five ids that the cohort does have.

Value

A Subject built from the matching row.

See Also

Subject, `subject_new()`, `cohort_new()`

Examples

```
data(example_cohort)
rat1 <- subject(example_cohort, "RAT001")
rat1@species
rat1@sex
```

`subjects` *Read the subject table of a cohort*

Description

A thin, named accessor for `cohort@subject_tbl`. Prefer it over the `@` operator in pipelines and scripts, so the accessor is the one place that would change if the underlying property ever did.

Usage

```
subjects(cohort)
```

Arguments

`cohort`	A Cohort object.

Value

`cohort@subject_tbl` as a tibble.

See Also

`samples()`, `completeness()`, `subject()`

Examples

```
data(example_cohort)
subjects(example_cohort)
```

`subject_new` *Create a Subject object*

Description

Constructs a Subject object representing an individual animal or biological sample in a study. Subjects must have a unique identifier and a species. All other attributes are optional.

Usage

```
subject_new(
  subject_id,
  species,
  sex = NA_character_,
  strain = NA_character_,
  genotype = NA_character_,
  cohort = NA_character_,
  timepoint = NA_character_,
  notes = NA_character_
)
```

Arguments

`subject_id`	Character scalar providing a unique identifier for the subject. Must be at least 1 character long.
`species`	Character scalar naming the species (e.g., "rat", "mouse", "human", "zebrafish"). Any value is allowed; it is stored lower-cased so that "Rat" and "rat" are the same species. Required.
`sex`	Character scalar indicating biological sex (e.g., "M", "F"). Optional and defaults to NA.
`strain`	Character scalar for strain or breed designation. Optional and defaults to NA.
`genotype`	Character scalar describing the genetic background or modification (e.g., "WT", "KO"). Optional and defaults to NA.
`cohort`	Character scalar for cohort membership or treatment group. Optional and defaults to NA.
`timepoint`	Character scalar indicating study timepoint or collection date. Optional and defaults to NA.
`notes`	Character scalar for additional metadata or observations. Optional and defaults to NA.

Details

Subject objects are S7 classes for storing individual-level metadata in cross-species studies. The design is species-agnostic: `species` is a free-form value, lower-cased so that a study can group subjects by species without also matching on case. Individual subjects are typically read from a Cohort with `subject()`.

Value

A Subject object with the species stored lower-cased.

See Also

Cohort for managing groups of subjects

Examples

```
# Create a rat subject
rat_subject <- subject_new(
  subject_id = "RAT001",
  species = "rat",
  sex = "M",
  strain = "Lewis",
  genotype = "WT",
  cohort = "Control"
)
print(rat_subject)

# Create a mouse subject
mouse_subject <- subject_new(
  subject_id = "MOUSE001",
  species = "mouse",
  sex = "F",
  strain = "C57BL/6",
  genotype = "KO"
)
print(mouse_subject)
```

---

translate *Translate features (or a whole cohort) across species or assemblies*

Description

The one entry point for cross-species translation. Coordinate features (variants, peaks, intervals) route to liftover, and gene-level features route to ortholog mapping. Given a Cohort, it translates every registered analysis according to its AnalysisSpec and returns a new, target-species cohort.

Usage

```
translate(x, to, from = NULL, ...)
```

Arguments

x	The thing to translate. Either: • a data.frame/tibble of features, or • a Cohort object.
to	Character scalar naming the target species/assembly.
from	For a feature table, a character scalar naming the source species/assembly; required for the "ortholog" strategy, optional (recorded as provenance) for "liftover". For a Cohort, NULL (default) infers the source species from subject_tbl$species: used directly when the cohort has one species, or resolved per analysis (and, for an analysis with a subject_id column, per subject) when it has more than one. Give it explicitly to override inference.
...	Strategy-specific arguments. For a **feature table**: • strategy: "liftover" (coordinate features; see liftover_intervals()) or "ortholog" (gene features; see ortholog_genes()). • chain: chain-file path for "liftover". • backend: translation backend (defaults: "rtracklayer" for liftover, "babelgene" for ortholog). • plus backend arguments such as gene_col/id_type for orthologs. For a **Cohort**: • chain: chain-file path used for any feature_type = "interval" analysis. A cohort translated from more than one source species can pass a named list instead, one chain per source species (e.g. list(rat = "rn7ToHg38.chain", mouse = "mm39ToHg38.chain")). • liftover_backend, ortholog_backend: backends for the two feature kinds. • analyses: optional character vector restricting which analyses to translate (defaults to all that have a registered spec with a feature_type).

Details

This function is **experimental** while the API settles.

Cohort-level translation keeps subjects and the sample map unchanged (the same biological subjects, viewed in another species' coordinate/gene space) and re-expresses each analysis's feature table. Analyses without a registered AnalysisSpec or without a feature_type are skipped with a warning rather than guessed at.

When the source species is inferred per subject, the combined TranslationResult for that analysis carries from as a vector (one entry per source species involved) and adds a .source_species column to mapped and unmapped, so a row's original species is never lost.

Value

A TranslationResult (feature table input) or a new Cohort whose analyses are expressed in to (Cohort input). For a cohort, per-analysis TranslationResults (including unmapped features) are retrievable with translation_report().

See Also

liftover_intervals(), ortholog_genes(), translation_report(), TranslationResult

Examples

```
# Feature-table input ------------------------------------------------
ints <- data.frame(seqnames = "chr1", start = 100, end = 200)
backend <- function(intervals, chain, ...) {
  list(
    mapped = tibble::tibble(
      .feature_id = intervals$.feature_id,
      seqnames = "chrT", start = 1L, end = 100L, strand = "*"
    ),
    unmapped = intervals[0, , drop = FALSE]
  )
}
translate(
  ints,
  to = "human", from = "rat",
  strategy = "liftover", chain = "none", backend = backend
)

# Cohort input -------------------------------------------------------
# One registered gene-level analysis, translated by an in-memory backend.
# The source species is inferred from the cohort's subjects.
manifest <- data.frame(
  subject_id = c("S1", "S2"), species = "rat", assay = "rna",
  sample_id = c("R1", "R2"), role = "tumor"
)
parsed <- validate_manifest(manifest)
cohort <- cohort_new(
  parsed$subject_tbl, parsed$sample_map,
  analyses = list(expr = data.frame(gene = c("Tp53", "Myc"), value = c(1, 2)))
)
spec <- analysis_spec_new(
  name = "expr", assay = "rna", level = "subject",
  feature_type = "gene", gene_col = "gene", id_type = "symbol"
)
cohort <- analysis_register(cohort, spec)
to_upper <- function(features, from, to, gene_col, id_type, ...) {
  mapped <- features
  mapped$ortholog <- toupper(mapped[[gene_col]])
  list(mapped = mapped, unmapped = features[0, , drop = FALSE])
}
human <- translate(cohort, to = "human", ortholog_backend = to_upper)
human@analyses$expr
```

`TranslationResult` *Result of a cross-species or cross-assembly translation*

Description

An S7 class holding the outcome of translating coordinate features or gene-level features from one species or assembly to another. It keeps the successfully translated features, the features that failed to map, and provenance so that translation is never lossy *silently*.

Usage

```
TranslationResult(
  mapped = NULL,
  unmapped = NULL,
  from = NA_character_,
  to = NA_character_,
  backend = NA_character_,
  stats = list()
)
```

Arguments

`mapped`	A tibble of successfully translated features. Includes a `.feature_id` column linking each output row back to its input row; one input may yield multiple output rows (multi-mapping).
`unmapped`	A tibble of input features that produced no output.
`from`	Character scalar naming the source species/assembly. Optional.
`to`	Character scalar naming the target species/assembly. Optional.
`backend`	Character scalar naming the translation backend used. Optional.
`stats`	Named list of summary counts (`n_input`, `n_mapped`, `n_unmapped`, `n_multi`).

Details

Construct these via `liftover_intervals()` or `translate()` rather than directly. Access the pieces with `result@mapped`, `result@unmapped`, and `translation_stats()`.

Value

A `TranslationResult` object with the given properties.

See Also

`liftover_intervals()`, `translate()`, `translation_stats()`

Examples

```
# Built by hand here to show the shape; translate() builds them for you.
res <- TranslationResult(
  mapped = tibble::tibble(.feature_id = 1L, gene = "Tp53", ortholog = "TP53"),
  unmapped = tibble::tibble(.feature_id = 2L, gene = "Gm12345"),
  from = "rat",
  to = "human",
```

```
  backend = "by_hand",
  stats = list(n_input = 2L, n_mapped = 1L, n_unmapped = 1L, n_multi = 0L)
)
res@unmapped
translation_stats(res)
```

`translation_report` *Retrieve per-analysis translation results from a cohort*

Description

After `translate()` has translated a Cohort, this returns the per-analysis TranslationResult objects (including the unmapped features and mapping statistics) recorded during translation.

Usage

```
translation_report(cohort)
```

Arguments

`cohort`	A Cohort produced by `translate()`.

Value

A named list with `from`, `to`, and `results` (a named list of TranslationResult objects, one per translated analysis), or `NULL` if the cohort has not been translated.

See Also

`translate()`, TranslationResult

Examples

```
# A cohort with one gene-level analysis, translated by an in-memory backend.
manifest <- data.frame(
  subject_id = "S1", species = "rat", assay = "rna", sample_id = "R1"
)
parsed <- validate_manifest(manifest)
cohort <- cohort_new(
  parsed$subject_tbl, parsed$sample_map,
  analyses = list(expr = data.frame(gene = c("Tp53", "Myc")))
)
spec <- analysis_spec_new(
  name = "expr", assay = "rna", level = "subject",
  feature_type = "gene", gene_col = "gene", id_type = "symbol"
)
cohort <- analysis_register(cohort, spec)
to_upper <- function(features, from, to, gene_col, id_type, ...) {
  mapped <- features
```

```
  mapped$ortholog <- toupper(mapped[[gene_col]])
  list(mapped = mapped, unmapped = features[0, , drop = FALSE])
}
human <- translate(cohort, to = "human", ortholog_backend = to_upper)

report <- translation_report(human)
report$from
report$to
translation_stats(report$results$expr)

# NULL for a cohort that has not been translated
translation_report(cohort)
```

`translation_stats` *Summary statistics for a translation*

Description

Returns the per-translation summary counts as a one-row tibble: number of input features, how many mapped, how many failed, and how many mapped to more than one location.

Usage

```
translation_stats(x)
```

Arguments

x	A TranslationResult object.

Value

A one-row tibble with columns `from`, `to`, `backend`, `n_input`, `n_mapped`, `n_unmapped`, `n_multi`, and `prop_mapped`.

See Also

TranslationResult

Examples

```
ints <- data.frame(
  seqnames = c("chr1", "chr1"),
  start = c(100, 5000),
  end = c(200, 5100)
)
# Using a trivial in-memory backend for illustration:
backend <- function(intervals, chain, ...) {
  list(
    mapped = tibble::tibble(
```

```
      .feature_id = intervals$.feature_id[1],
      seqnames = "chrT", start = 1L, end = 100L, strand = "*"
    ),
    unmapped = intervals[-1, , drop = FALSE]
  )
}
res <- liftover_intervals(
  ints, chain = "none", to = "human", backend = backend
)
translation_stats(res)
```

`validate_cohort` *Validate a Cohort object*

Description

Checks the subject table, the sample map, and the link between them. `cohort_new()` calls this function after it builds the object. Call it directly to check a cohort that was built or changed by other means.

Usage

```
validate_cohort(x)
```

Arguments

`x`	A Cohort object.

Details

The checks are:

- `x` is a Cohort object.
- `subject_tbl` and `sample_map` are data frames.
- `subject_tbl` has the columns `subject_id` and `species`, both character, with no missing value. An empty string counts as missing. `species` is a free-form value; any organism is allowed.
- `subject_tbl$subject_id` has no duplicate.
- `sample_map` has the columns `subject_id`, `assay`, `sample_id`, and `role`, all character. The first three have no missing value.
- Every `sample_map$subject_id` exists in `subject_tbl`.

Value

`TRUE`, invisibly, when the cohort is valid. Otherwise an error that lists every problem found.

See Also

cohort_new() for Cohort construction, validate_manifest() for manifest validation

Examples

```
subjects <- data.frame(
  subject_id = "RAT001",
  species = "rat",
  sex = "M"
)
samples <- data.frame(
  subject_id = "RAT001",
  assay = "wes",
  sample_id = "WES_R001",
  role = "tumor"
)
cohort <- cohort_new(
  subject_tbl = subjects,
  sample_map = samples
)

# cohort_new() already ran the checks. Run them again by hand.
validate_cohort(cohort)
```

---

validate_manifest *Validate and structure a long-format sample manifest*

Description

Validates a tidy, long-format manifest (one row per sample) and structures it into a subject metadata table and a canonical long-format sample map. The design is deliberately species- and assay-agnostic: any organism and any assay (WGS, WES, ATAC-seq, bulk RNA, single-cell, ...) are represented as values, never as bespoke columns or per-assay tables.

Usage

```
validate_manifest(
  manifest,
  sample_cols = NULL,
  species = NULL,
  allow_duplicates = FALSE
)
```

Arguments

manifest	A data.frame or tibble in **long format**, one row per sample. Required columns:

- subject_id (character; coerced): subject the sample belongs to.

- `assay` (character; coerced): assay type, e.g. `"wgs"`, `"wes"`, `"atac"`, `"bulk_rna"`, `"scrna"`.
- `sample_id` (character; coerced): unique sample identifier.

Optional sample-level columns:

- `role` (character): role of the sample within its assay, e.g. `"tumor"`, `"normal"`. Defaults to `NA` when absent.
- A column named in `sample_cols`, or recognized by name (see Details).

Any remaining columns (e.g. `species`, `sex`, `strain`, `genotype`, `cohort`, `timepoint`, `notes`) are treated as **subject-level metadata**, coerced to character, and must be constant within a `subject_id`.

`sample_cols` Optional character vector naming additional columns to keep at the sample level (in `sample_map`) rather than treat as subject-level metadata. Use it for a column that varies per sample but is not one of the columns `validate_manifest()` already recognizes.

`species` Optional character scalar. When `manifest` has no `species` column, this value fills one. Ignored when `manifest` already has a `species` column. Defaults to `NULL` (no column added).

`allow_duplicates`
Logical. If `FALSE` (default), a repeated `sample_id` raises an error. If `TRUE`, duplicates are kept.

Details

Every column is coerced to character, so a numeric, logical, or factor column never reaches `cohort_new()` in a form that would fail there. Empty strings are treated as missing. Every sample row must carry a non-missing `subject_id`, `assay`, and `sample_id`.

Beyond the four canonical columns, these names are always kept at the sample level when present: `specimen_id`, `library_id`, `vendor_id`, `replicate`, `lane`, `run`, `flowcell`, `strandedness`, `fastq_1`, `fastq_2`, `bam`, `cram`, `vcf`, `matrix_dir`, `h5`, `qc_status`, and `qc_reason`. Add any other column that varies per sample with `sample_cols`; a column that varies within a subject but is not recognized or declared raises the conflicting-metadata error below.

`species`, when present (in the manifest or filled from the `species` argument), is lower-cased so that `"Rat"` and `"rat"` are the same subject-level value. Any species value is allowed; the manifest layer does not restrict it to a fixed list of organisms.

`sample_id` must be unique across the whole manifest, not only within a subject or assay, unless `allow_duplicates = TRUE`.

Per-assay wide views (e.g. tumor/normal pairs) are not part of the core contract; derive them on demand from `sample_map` with `sample_pairs()`.

Value

A list with three elements:

- `subject_tbl`: Tibble with one row per `subject_id` containing the subject-level metadata columns, all character.

- sample_map: Canonical long-format tibble with columns subject_id, assay, sample_id, role, and any recognized or declared extra sample-level columns, all character.
- completeness_tbl: Tibble with one row per subject_id x assay summarizing the number of samples (n_samples).

See Also

read_manifest_csv() for reading a manifest from CSV, cohort_new() for creating a Cohort from manifest data

Examples

```
manifest <- data.frame(
  subject_id = c("RAT001", "RAT001", "MOUSE1", "HUM01"),
  species = c("rat", "rat", "mouse", "human"),
  assay = c("wes", "wes", "atac", "wgs"),
  sample_id = c("WES_T1", "WES_N1", "ATAC_1", "WGS_T1"),
  role = c("tumor", "normal", NA, "tumor"),
  fastq_1 = c("t1_R1.fq.gz", "n1_R1.fq.gz", "a1_R1.fq.gz", "g1_R1.fq.gz"),
  stringsAsFactors = FALSE
)

parsed <- validate_manifest(manifest)
parsed$subject_tbl
parsed$sample_map
parsed$completeness_tbl
```

---

write_manifest *Write a manifest or a cohort's tables to a delimited file*

Description

Joins a cohort's (or a validate_manifest() result's) sample_map and subject_tbl back into one long-format manifest and writes it to a CSV, TSV, or other delimited file.

Usage

```
write_manifest(x, path, delim = NULL)
```

Arguments

x	A Cohort object, or the list returned by validate_manifest() or read_manifest() (anything with subject_tbl and sample_map).
path	Output file path. The delimiter is chosen from the extension unless delim is given.
delim	Optional character scalar overriding delimiter detection.

Details

Columns are ordered `subject_id`, then the other subject-level columns, then the sample-level columns (`assay`, `sample_id`, `role`, and any extra ones). A missing value is written as an empty field.

Value

The written manifest tibble, invisibly.

See Also

`read_manifest()`, `validate_manifest()`, `cohort_save()`

Examples

```
manifest <- data.frame(
  subject_id = "R1",
  species = "rat",
  assay = "wes",
  sample_id = "T1",
  role = "tumor",
  stringsAsFactors = FALSE
)
parsed <- validate_manifest(manifest)

out <- tempfile(fileext = ".csv")
write_manifest(parsed, out)
readLines(out)
```

`write_study_yaml` *Write a cohort as a study YAML file*

Description

The inverse of `read_study_yaml()`: writes the cohort's study metadata, its manifest, its paths, and its registered analysis specs to a study YAML file and a manifest file alongside it.

Usage

```
write_study_yaml(cohort, path, manifest = "manifest.csv")
```

Arguments

`cohort`	A Cohort object.
`path`	Output path for the study YAML file.
`manifest`	Path for the manifest file, resolved relative to `path`'s directory unless absolute. Default `"manifest.csv"`.

Details

A cohort has no stored corrections file, so a `corrections:` key is never written; the manifest written out already reflects any correction that was applied before the cohort was built.

Value

`path`, invisibly.

See Also

`read_study_yaml()`, `write_manifest()`

Examples

```
data(example_cohort)
dir <- tempfile()
dir.create(dir)
write_study_yaml(example_cohort, file.path(dir, "study.yaml"))
cat(readLines(file.path(dir, "study.yaml")), sep = "\n")
```

Index

∗ **datasets**
example_cohort, 33

analysis_files, 8
analysis_files(), *41, 42*
analysis_list, 9
analysis_list(), *11, 12*
analysis_register, 10
analysis_register(), *4*, *7–9*, *12*, *15*, *20*, *22*, *41*
analysis_spec, 12
analysis_spec(), *9*, *11*
analysis_spec_new, 13
analysis_spec_new(), *4*, *6–8*, *11*, *56*
AnalysisSpec, 6, *15*, *41–43*, *72, 73*
apply_corrections, 16
apply_corrections(), *5*, *31*, *51, 52*, *56*
as_coldata, 17
as_coldata(), *4*, *35*

babelgene::orthologs(), *47*
biocohort *(*biocohort-package*)*, 3
biocohort-package, 3

check_paths, 18
check_paths(), *4*, *63*
Cohort, *8*, *17, 18*, 19, *21–25*, *28–32*, *34*, *41–43*, *45*, *51*, *56*, *60*, *62*, *65*, *67*, *69, 70*, *72, 73*, *76*, *78*, *81, 82*
cohort_contrasts, 21
cohort_contrasts(), *4*, *25*
cohort_derive, 22
cohort_derive(), *4*, *20*, *32*
cohort_filter, 23
cohort_filter(), *4*, *20–22*, *27, 28*
cohort_groups, 25
cohort_groups(), *4*, *21*, *23*
cohort_new, 26
cohort_new(), *4*, *20*, *24*, *34*, *54–56*, *70*, *78–81*
cohort_qc, 27
cohort_qc(), *4*, *20*, *51*
cohort_read, 29
cohort_read(), *4*, *30*
cohort_save, 30
cohort_save(), *4*, *29*, *82*
completeness, 30
completeness(), *4*, *60*, *70*
corrections_log, 31
corrections_log(), *16*

derive_log, 32
derive_log(), *4*, *22, 23*
dplyr::filter(), *24*

ensure_dir, 32
ensure_dir(), *50*
example_cohort, 33

fs::dir_create(), *32*
fs::path(), *49*

join_metadata, 34
join_metadata(), *4*, *18*

liftover_backends, 35
liftover_backends(), *38*, *58*
liftover_crossmap, 36
liftover_crossmap(), *38, 39*, *41*, *58*
liftover_intervals, 37
liftover_intervals(), *5*, *36*, *39*, *41*, *46*, *57, 58*, *73–75*
liftover_rtracklayer, 39
liftover_rtracklayer(), *36*, *38*, *58*
liftover_vcf, 40
liftover_vcf(), *36*, *38*
load_analyses, 41
load_analyses(), *4*, *8*, *43*
load_analysis, 42
load_analysis(), *4*, *7*, *14*, *41, 42*

manifest_from_wide, 44

manifest_from_wide(), *4*, *54*

ortholog_babelgene, 46
ortholog_babelgene(), *49*, *59*
ortholog_backends, 47
ortholog_backends(), *48*, *59*
ortholog_genes, 48
ortholog_genes(), *5*, *46–48*, *58*, *59*, *73*, *74*
orthologize, 45

project_path, 49
project_path(), *50*
project_root, 50
project_root(), *49*, *50*, *52*

qc_log, 51
qc_log(), *4*, *22*, *27*, *28*

read_corrections, 51
read_corrections(), *5*, *16*
read_dotenv, 52
read_manifest, 53
read_manifest(), *4*, *45*, *55*, *56*, *81*, *82*
read_manifest_csv, 54
read_manifest_csv(), *20*, *34*, *81*
read_study_yaml, 56
read_study_yaml(), *5*, *82*, *83*
readr::read_csv(), *55*
readr::read_delim(), *53*
readxl::read_excel(), *53*, *54*
register_liftover_backend, 57
register_liftover_backend(), *5*, *36*
register_ortholog_backend, 58
register_ortholog_backend(), *5*, *48*
rtracklayer::liftOver(), *39*

sample_pairs, 60
sample_pairs(), *4*, *6*, *7*, *14*, *43*, *60*, *80*
sample_sheet, 62
sample_sheet(), *4*, *18*, *64*
sample_sheet_templates, 64
sample_sheet_templates(), *62*, *63*
samples, 59
samples(), *4*, *18*, *24*, *31*, *35*, *63*, *70*
saveRDS(), *30*
stats::relevel(), *17*
Study, *27*, 64
study_new, 66
study_new(), *4*, *56*, *65*
Subject, *20*, 68, *69*, *70*
subject, 69
subject(), *4*, *20*, *26*, *27*, *33*, *34*, *68*, *70*, *71*
subject_new, 71
subject_new(), *68–70*
subjects, 70
subjects(), *4*, *24*, *25*, *31*, *60*
Sys.setenv(), *52*

translate, 72
translate(), *5*, *7*, *14*, *38*, *41–43*, *45*, *46*, *49*, *75*, *76*
translation_report, 76
translation_report(), *73*, *74*
translation_stats, 77
translation_stats(), *38*, *49*, *75*
TranslationResult, *37*, *38*, *41*, *48*, *49*, *73*, *74*, 74, *76*, *77*

validate_cohort, 78
validate_cohort(), *19*, *20*, *26*, *27*
validate_manifest, 79
validate_manifest(), *4*, *20*, *26*, *27*, *31*, *34*, *44*, *45*, *53–56*, *61*, *79–82*

write_manifest, 81
write_manifest(), *4*, *30*, *83*
write_study_yaml, 82
write_study_yaml(), *5*, *56*