Package 'diseasystore'

November 16, 2026

Title Feature Stores for the 'diseasy' Framework

Version 0.4.1

Description Simple feature stores and tools for creating personalised feature stores.
'diseasystore' powers feature stores which can automatically link and aggregate features to a given stratification
level. These feature stores are automatically time-
versioned (powered by the 'SCDB' package) and allows you to easily
and dynamically compute features as part of your continuous integration.

License GPL (>= 3)

Encoding UTF-8

Language en-GB

Depends R (>= 4.3.0)

Imports checkmate, curl, DBI, dbplyr, dplyr, glue, ISOweek, jsonlite, lubridate, pkgcond, purrr, readr, rlang, R6, SCDB (>= 0.5.2), stringr, tidyr, tidyselect

Suggests devtools, duckdb, ggplot2, here, knitr, lintr, microbenchmark, odbc (>= 1.6.4), pkgdown, rmarkdown, RSQLite, RPostgres, testthat (>= 3.0.0), tibble, spelling, usethis, withr

VignetteBuilder knitr

URL https://github.com/ssi-dk/diseasystore, https://ssi-dk.github.io/diseasystore/

BugReports https://github.com/ssi-dk/diseasystore/issues

Config/testthat/edition 3

LazyData true

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Rasmus Skytte Randløv [aut, cre] (ORCID: <https://orcid.org/0000-0002-5860-3838>), Kaare Græsbøll [rev] (ORCID: <https://orcid.org/0000-0002-6258-8212>),

Kasper Schou Telkamp [rev] (ORCID:
<https://orcid.org/0009-0001-5126-0190>),
Lasse Engbo Christiansen [rev] (ORCID:
<https://orcid.org/0000-0001-5019-1931>),
Marcus Munch Grünewald [rev] (ORCID:
<https://orcid.org/0009-0006-8090-406X>),
Sofia Myrup Otero [rev],
Statens Serum Institut, SSI [cph, fnd]

Maintainer Rasmus Skytte Randløv <rske@ssi.dk>

Repository CRAN

Date/Publication 2026-11-16 15:30:02 UTC

Contents

add_years . . . 2
age_labels . . . 3
age_on_date . . . 4
aggregators . . . 4
available_diseasystores . . . 6
diseasyoption . . . 6
DiseasystoreBase . . . 7
DiseasystoreEcdcRespiratoryViruses . . . 11
DiseasystoreGoogleCovid19 . . . 12
DiseasystoreSimulist . . . 13
diseasystore_exists . . . 14
drop_diseasystore . . . 14
FeatureHandler . . . 15
get_diseasystore . . . 17
source_conn_helpers . . . 18
test_diseasystore . . . 19
to_diseasystore_case . . . 20
%.% . . . 21

Index **22**

`add_years` *Backend-dependent time interval (in years)*

Description

Provides the sql code for a time interval (in years).

Usage

```
add_years(reference_date, years, conn)
```

Arguments

reference_date	(Date(1) or character(1)) The date to add years to (or name of column containing the reference date).
years	(numeric(1) or character(1)) The length of the time interval in whole years (or name of column containing the number of years).
conn	(DBIConnection or function) A database connection or function that opens a database connection.

Value

SQL query for the time interval.

Examples

```
conn <- SCDB::get_connection(drv = RSQLite::SQLite())

dplyr::copy_to(conn, data.frame(birth = as.Date("2001-04-03"), "test_age")) |>
  dplyr::mutate(first_birthday = !!add_years("birth", 1, conn))

DBI::dbDisconnect(conn)
```

---

age_labels *Provides sortable labels for age groups*

Description

Provides sortable labels for age groups

Usage

```
age_labels(age_cuts)
```

Arguments

age_cuts	(numeric()) The lower bound of the groups (0 is implicitly included).

Value

A vector of labels with zero-padded numerics so they can be sorted easily.

Examples

```
age_labels(c(5, 12, 20, 30))
```

age_on_date *Compute the age (in years) on a given date*

Description

Provides the sql code to compute the age of a person on a given date.

Usage

```
age_on_date(birth, reference_date, conn)
```

Arguments

`birth`	(`character(1)`) Name of the birth date column.
`reference_date`	(`Date(1)` or `character(1)`) The date to compute the age for (or name of column containing the reference date).
`conn`	(`DBIConnection` or `function`) A database connection or function that opens a database connection.

Value

SQL query that computes the age on the given date.

Examples

```
conn <- SCDB::get_connection(drv = RSQLite::SQLite())

dplyr::copy_to(conn, data.frame(birth = as.Date("2001-04-03"), "test_age")) |>
  dplyr::mutate(age = !!age_on_date("birth", as.Date("2024-02-28"), conn))

DBI::dbDisconnect(conn)
```

aggregators *title Feature aggregators*

Description

When designing a diseasystore (see vignette("extending-diseasystore")), each feature must specify how that data should be summarised when joined with other data.

The automatic coupling and aggregation is essentially a three part process: First, data for the observable is joined with data required to form the stratifications. Next, the combined data is grouped at the level of stratification requested. Finally, the data is summarised in each group to form a single value for the group.

Below, we provide a set of simple function that perform this final summarisation step.

For semi-aggregated data, a typical key_join aggregator is the key_join_sum() which just adds the observations across the group

In some cases, such as if the observable is the "maximum temperature" the relevant aggregator would be the key_join_max() aggregator which takes the max across the group.

When working with individual level data, the observables often consists of a record per individual and to summarise the data we instead need the key_join_count() aggregator to get the size of the group.

Usage

```
key_join_sum(.data, feature)

key_join_max(.data, feature)

key_join_min(.data, feature)

key_join_count(.data, feature)
```

Arguments

.data	(any) The data object to perform the operation on.
feature	(character) Name of the feature to perform the aggregation over

Value

A dplyr::summarise to aggregate the features together using the given function (sum/max/min/count)

Examples

```
# Primarily used within the framework but can be used individually:

data <- dplyr::mutate(mtcars, key_name = rownames(mtcars), .before = dplyr::everything())

key_join_sum(data, "mpg")    # sum(mtcars$mpg)
key_join_max(data, "mpg")    # max(mtcars$mpg)
key_join_min(data, "mpg")    # min(mtcars$mpg)
key_join_count(data, "mpg")  # nrow(mtcars)
```

available_diseasystores *Detect available diseasystores*

Description

Detect available diseasystores

Usage

```
available_diseasystores()
```

Value

The installed diseasystores on the search path

Examples

```
available_diseasystores()  # DiseasystoreGoogleCovid19 + more from other packages
```

diseasyoption *Helper function to get options related to diseasy*

Description

Helper function to get options related to diseasy

Usage

```
diseasyoption(option, class = NULL, namespace = NULL, .default = NULL)
```

Arguments

option	(character(1)) Name of the option to get.
class	(character(1) or R6::R6class Diseasy* instance) Either the classname or the object the option applies to.
namespace	(character(1)) The namespace of the option (e.g. "diseasy" or "diseasystore").
.default	(any) The default value to return if no option is set.

Value

- If `option` is given, the most specific option within the `diseasy` framework for the given option and class.
- If `option` is missing, all options related to `diseasy` packages.

Examples

```
# Retrieve default option for source conn
diseasyoption("source_conn")

# Retrieve DiseasystoreGoogleCovid19 specific option for source conn
diseasyoption("source_conn", "DiseasystoreGoogleCovid19")

# Try to retrieve specific option for source conn for a non existent / un-configured diseasystore
diseasyoption("source_conn", "DiseasystoreNonExistent") # Returns default source_conn

# Try to retrieve specific non-existent option
diseasyoption("non_existent", "DiseasystoreGoogleCovid19", .default = "Use this")
```

`DiseasystoreBase` *diseasystore base handler*

Description

This `DiseasystoreBase` R6 class forms the basis of all feature stores. It defines the primary methods of each feature stores as well as all of the public methods.

Value

A new instance of the `DiseasystoreBase` R6 class.

Active bindings

`ds_map (named list(character))`
: A list that maps features known by the feature store to the corresponding feature handlers that compute the features. Read only.

`available_features (character())`
: A list of available features in the feature store. Read only.

`available_observables (character())`
: A list of available observables in the feature store. Read only.

`available_stratifications (character())`
: A list of available stratifications in the feature store. Read only.

`observables_regex (character(1))`
: A list of available stratifications in the feature store. Read only.

`label (character(1))`
: A human readable label of the feature store. Read only.

`source_conn` (`DBIConnection` or `file path`)
: Used to specify where data is located. Read only. Can be `DBIConnection` or file path depending on the `diseasystore`.

`target_conn` (`DBIConnection`)
: A database connection to store the computed features in. Read only.

`target_schema` (`character`)
: The schema to place the feature store in. Read only. If the database backend does not support schema, the tables will be prefixed with `<target_schema>.`.

`start_date` (`Date`)
: Study period start. Read only.

`end_date` (`Date`)
: Study period end. Read only.

`min_start_date` (`Date`)
: (Minimum)Study period start. Read only.

`max_end_date` (`Date`)
: (Maximum)Study period end. Read only.

`slice_ts` (`Date` or `character`)
: Date or timestamp (parsable by `as.POSIXct`) to slice the (time-versioned) data on. Read only.

Methods

Public methods:

- `DiseasystoreBase$new()`
- `DiseasystoreBase$get_feature()`
- `DiseasystoreBase$key_join_features()`
- `DiseasystoreBase$determine_missing_ranges()`
- `DiseasystoreBase$clone()`

`DiseasystoreBase$new()`: Creates a new instance of the `DiseasystoreBase` R6 class.

Usage:

```
DiseasystoreBase$new(
  start_date = NULL,
  end_date = NULL,
  slice_ts = NULL,
  source_conn = NULL,
  target_conn = NULL,
  target_schema = NULL,
  verbose = diseasyoption("verbose", self)
)
```

Arguments:

`start_date` (`Date`)
: Study period start.

`end_date` (`Date`)
: Study period end.

`slice_ts` (`Date` or `character`)
: Date or timestamp (parsable by `as.POSIXct`) to slice the (time-versioned) data on.

`source_conn` (`DBIConnection` or `file path`)
: Used to specify where data is located. Can be `DBIConnection` or file path depending on the `diseasystore`.

`target_conn` (`DBIConnection`)
: A database connection to store the computed features in.

`target_schema` (`character`)
: The schema to place the feature store in. If the database backend does not support schema, the tables will be prefixed with `<target_schema>.`.

`verbose` (`boolean`)
: Boolean that controls enables debugging information.

Returns: A new instance of the `DiseasystoreBase` R6 class.

`DiseasystoreBase$get_feature()`: Computes, stores, and returns the requested feature for the study period.

Usage:

```
DiseasystoreBase$get_feature(
  feature,
  start_date = self %.% start_date,
  end_date = self %.% end_date,
  slice_ts = self %.% slice_ts
)
```

Arguments:

`feature` (`character`)
: The name of a feature defined in the feature store.

`start_date` (`Date`)
: Study period start.

`end_date` (`Date`)
: Study period end.

`slice_ts` (`Date` or `character`)
: Date or timestamp (parsable by `as.POSIXct`) to slice the (time-versioned) data on.

Returns: A tbl_dbi with the requested feature for the study period.

`DiseasystoreBase$key_join_features()`: Joins various features from the feature store assuming a primary feature (observable) that contains keys to witch the secondary features (defined by `stratification`) are joined.

Usage:

```
DiseasystoreBase$key_join_features(
  observable,
  stratification = NULL,
  start_date = self %.% start_date,
  end_date = self %.% end_date
)
```

Arguments:

observable (character)
: The observable to provide data or prediction for.

stratification (list(quosures) or NULL)
: Use rlang::quos(...) to specify stratification. If given, expressions in stratification evaluated to give the stratification level.

start_date (Date)
: Study period start.

end_date (Date)
: Study period end.

Returns: A tbl_dbi with the requested joined features for the study period.

`DiseasystoreBase$determine_missing_ranges()`: Determine which dates are not already computed.

This method parses the `message` fields of the logs related to `target_table` on the date given by `slice_ts`. These messages contains information on the date-ranges that are already computed. Once parsed, the method compares the computed date-ranges with the requested date-range.

Usage:

```
DiseasystoreBase$determine_missing_ranges(
  target_table,
  start_date,
  end_date,
  slice_ts
)
```

Arguments:

target_table (character or Id)
: The feature table to investigate.

start_date (Date)
: Study period start.

end_date (Date)
: Study period end.

slice_ts (Date or character)
: Date or timestamp (parsable by as.POSIXct) to slice the (time-versioned) data on.

Returns: (tibble)
A data.frame containing continuous date-ranges that require computation to cover the requested period

`DiseasystoreBase$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DiseasystoreBase$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# DiseasystoreBase is mostly used as the basis of other, more specific, classes
# The DiseasystoreBase can be initialised individually if needed.

ds <- DiseasystoreBase$new(source_conn = NULL,
                           target_conn = DBI::dbConnect(RSQLite::SQLite()))

rm(ds)
```

`DiseasystoreEcdcRespiratoryViruses`
feature store handler of EU-ECDC Respiratory viruses features

---

Description

This `DiseasystoreEcdcRespiratoryViruses` R6 brings support for using the EU-ECDC Respiratory viruses weekly data repository. See the vignette("diseasystore-ecdc-respiratory-viruses") for details on how to configure the feature store.

Value

A new instance of the `DiseasystoreEcdcRespiratoryViruses` R6 class.

Super class

`DiseasystoreBase` -> `DiseasystoreEcdcRespiratoryViruses`

Methods

Public methods:

- `DiseasystoreEcdcRespiratoryViruses$new()`
- `DiseasystoreEcdcRespiratoryViruses$clone()`

`DiseasystoreEcdcRespiratoryViruses$new()`**:** Creates a new instance of the `DiseasystoreEcdcRespiratoryViruse` R6 class.

Usage:

```
DiseasystoreEcdcRespiratoryViruses$new(...)
```

Arguments:

`...` Arguments passed to the ?`DiseasystoreBase` constructor.

Returns: A new instance of the `DiseasystoreEcdcRespiratoryViruses` R6 class.

`DiseasystoreEcdcRespiratoryViruses$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DiseasystoreEcdcRespiratoryViruses$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
ds <- DiseasystoreEcdcRespiratoryViruses$new(
  source_conn = ".",
  target_conn = DBI::dbConnect(RSQLite::SQLite())
)


rm(ds)
```

DiseasystoreGoogleCovid19 *feature store handler of Google Health COVID-19 Open Data features*

Description

This DiseasystoreGoogleCovid19 R6 brings support for using the Google Health COVID-19 Open Data repository. See the vignette("diseasystore-google-covid-19") for details on how to configure the feature store.

Value

A new instance of the DiseasystoreGoogleCovid19 R6 class.

Super class

DiseasystoreBase -> DiseasystoreGoogleCovid19

Methods

Public methods:

- DiseasystoreGoogleCovid19$clone()

DiseasystoreGoogleCovid19$clone(): The objects of this class are cloneable with this method.

Usage:

```
DiseasystoreGoogleCovid19$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
ds <- DiseasystoreGoogleCovid19$new(
  source_conn = ".",
  target_conn = DBI::dbConnect(RSQLite::SQLite())
)


rm(ds)
```

DiseasystoreSimulist *feature store handler of synthetic* simulist *features*

Description

This DiseasystoreSimulist R6 brings support for individual level data.

Value

A new instance of the DiseasystoreSimulist R6 class.

Super class

DiseasystoreBase -> DiseasystoreSimulist

Methods

Public methods:

- DiseasystoreSimulist$new()
- DiseasystoreSimulist$clone()

DiseasystoreSimulist$new(): Creates a new instance of the DiseasystoreSimulist R6 class.

Usage:

```
DiseasystoreSimulist$new(...)
```

Arguments:

... Arguments passed to the ?DiseasystoreBase constructor.

Returns: A new instance of the DiseasystoreSimulist R6 class.

DiseasystoreSimulist$clone(): The objects of this class are cloneable with this method.

Usage:

```
DiseasystoreSimulist$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
ds <- DiseasystoreSimulist$new(
  source_conn = ".",
  target_conn = DBI::dbConnect(duckdb::duckdb())
)


rm(ds)
```

diseasystore_exists *Check for the existence of a* diseasystore *for the case definition*

Description

Check for the existence of a diseasystore for the case definition

Usage

```
diseasystore_exists(label)
```

Arguments

label	(character) A character string that controls which feature store to get data from.

Value

TRUE if the given diseasystore can be matched to a diseasystore on the search path. FALSE otherwise.

Examples

```
diseasystore_exists("Google COVID-19")  # TRUE
diseasystore_exists("Non existent diseasystore")  # FALSE
```

drop_diseasystore *Drop feature stores from DB*

Description

Drop feature stores from DB

Usage

```
drop_diseasystore(
  pattern = NULL,
  schema = diseasyoption("target_schema", namespace = "diseasystore"),
  conn = SCDB::get_connection()
)
```

Arguments

pattern	(character(1)) Pattern to match the tables by
schema	(character) A database schema. If the database backend does not support schema, the tables will be prefixed with <schema>. The location where the diseasystore stores data.
conn	(DBIConnection or function) A database connection or function that opens a database connection.

Value

NULL (called for side effects)

Examples

```
conn <- SCDB::get_connection(drv = RSQLite::SQLite())

drop_diseasystore(conn = conn)

DBI::dbDisconnect(conn)
```

FeatureHandler *FeatureHandler*

Description

This FeatureHandler R6 handles individual features for the feature stores. They define the three methods associated with features (compute, get and key_join).

Value

A new instance of the FeatureHandler R6 class.

Active bindings

compute (function)
: A function of the form "function(start_date, end_date, slice_ts, source_conn, ds (optional), ...)". This function should compute the feature from the source connection.

get (function)
: A function of the form "function(target_table, slice_ts, target_conn)". This function should retrieve the computed feature from the target connection.

key_join (function)
: One of the aggregators from aggregators.

Methods

Public methods:

- FeatureHandler$new()
- FeatureHandler$clone()

`FeatureHandler$new()`: Creates a new instance of the `FeatureHandler` R6 class.

Usage:

```
FeatureHandler$new(compute = NULL, get = NULL, key_join = NULL)
```

Arguments:

`compute (function)`
A function of the form "function(start_date, end_date, slice_ts, source_conn, ds (optional), ...)".
This function should return a `data.frame` with the computed feature (computed from the source connection). The `data.frame` should contain the following columns:

- key_*: One (or more) columns containing keys to link this feature with other features
- *: One (or more) columns containing the features that are computed
- valid_from, valid_until: A set of columns containing the time period for which this feature information is valid.

`get (function)`
(Optional). A function of the form "function(target_table, slice_ts, target_conn, ...)". This function should retrieve the computed feature from the target connection.

`key_join (function)`
A function like one of the aggregators from aggregators().
The function should return an expression on the form: dplyr::summarise(.data, dplyr::across(.cols = tidyselect::all_of(feature), .fns = list(n = ~ aggregation function), .names = "{.fn}"), .groups = "drop")

Returns: A new instance of the `FeatureHandler` R6 class.

`FeatureHandler$clone()`: The objects of this class are cloneable with this method.

Usage:

```
FeatureHandler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# The FeatureHandler is typically configured as part of making a new Diseasystore.
# Most often, we need only specify `compute` and `key_join` to get a functioning FeatureHandler

# In this example we use mtcars as the basis for our features
conn <- SCDB::get_connection(drv = RSQLite::SQLite())

# We use mtcars as our basis. First we add the rownames as an actual column
```

```
data <- dplyr::mutate(mtcars, key_name = rownames(mtcars), .before = dplyr::everything())

# Then we add some imaginary times where these cars were produced
data <- dplyr::mutate(data,
              production_start = as.Date(Sys.Date()) + floor(runif(nrow(mtcars)) * 100),
                 production_end  = production_start + floor(runif(nrow(mtcars)) * 365))

dplyr::copy_to(conn, data, "mtcars")

# In this example, the feature we want is the "maximum miles per gallon"
# The feature in question in the mtcars data set is then "mpg" and when we need to reduce
# our data set, we want to use the "max()" function.

# We first write a compute function for the mpg in our modified mtcars data set
# Our goal is to get the mpg of all cars that were in production at the between start/end_date
compute_mpg <- function(start_date, end_date, slice_ts, source_conn) {
  out <- SCDB::get_table(source_conn, "mtcars", slice_ts = slice_ts) |>
    dplyr::filter({{ start_date }} <= .data$production_end,
                  .data$production_start <= {{ end_date }}) |>
    dplyr::transmute("key_name", "mpg",
                     "valid_from" = "production_start",
                     "valid_until" = "production_end")

  return(out)
}

# We can now combine into our FeatureHandler
fh_max_mpg <- FeatureHandler$new(compute = compute_mpg, key_join = key_join_max)

DBI::dbDisconnect(conn)
```

`get_diseasystore` *Get the* `diseasystore` *for the case definition*

Description

Get the `diseasystore` for the case definition

Usage

```
get_diseasystore(label)
```

Arguments

`label`	(`character`) A character string that controls which feature store to get data from.

Value

The diseasystore generator for the diseasystore matching the given label

Examples

```
ds <- get_diseasystore("Google COVID-19") # Returns the DiseasystoreGoogleCovid19 generator
```

`source_conn_helpers` *File path helper for different source_conn*

Description

- source_conn_path: static url / directory. This helper determines whether source_conn is a file path or URL and creates the full path to the the file as needed based on the type of source_conn.
- source_conn_github: static GitHub API url / git directory. This helper determines whether source_conn is a git directory or a GitHub API creates the full path to the the file as needed based on the type of source_conn.

  A GitHub token can be configured in the "GITHUB_PAT" environment variable to avoid rate limiting.

  If the basename of the requested file contains a date, the function will use fuzzy-matching to determine the closest matching, chronologically earlier, file location to return.

Usage

```
source_conn_path(source_conn, file)

source_conn_github(source_conn, file, pull = TRUE)
```

Arguments

`source_conn`	(`character(1)`) # nolint: documentation_template_linter File location (path or URL).
`file`	(`character(1)`) Name (including path) of the file at the location.
`pull`	(`logical(1)`) Should "git pull" be called on the local repository before reading files?

Value

(`character(1)`)
The full path to the requested file.

Examples

```
# Simulating a data directory
source_conn <- "data_dir"
dir.create(source_conn)
write.csv(mtcars, file.path(source_conn, "mtcars.csv"))
write.csv(iris, file.path(source_conn, "iris.csv"))
```

```
# Get file path for mtcars.csv
source_conn_path(source_conn, "mtcars.csv")

# Clean up
unlink(source_conn, recursive = TRUE)
```

`test_diseasystore` *Test a given diseasy store*

Description

This function runs a battery of tests of the given diseasystore.

The supplied diseasystore must be a generator for the diseasystore, not an instance of the diseasystore.

The tests assume that data has been made available locally to run the majority of the tests. The location of the local data should be configured in the options for "source_conn" of the given diseasystore before calling test_diseasystore.

Usage

```
test_diseasystore(
  diseasystore_generator = NULL,
  conn_generator = NULL,
  data_files = NULL,
  target_schema = "test_ds",
  test_start_date = NULL,
  skip_backends = NULL,
  ...
)
```

Arguments

`diseasystore_generator`
: (`Diseasystore*`)
The diseasystore R6 class generator to test.

`conn_generator` (`function`)
: Function that generates a `list()` of connections use as target_conn. Should take a `skip_backend` that does not open connections for the given backends.

`data_files` (`character()`)
: List of files that should be available when testing.

`target_schema` (`character`)
: The schema to place the feature store in. If the database backend does not support schema, the tables will be prefixed with `<target_schema>.`.

`test_start_date`
: (`Date`)
The earliest date to retrieve data from during tests.

skip_backends (character())
: List of connection types to skip tests for due to missing functionality.

... : Other parameters passed to the diseasystore generator.

Value

NULL (called for side effects)

Examples

```
# Generator for connections
conns <- \(skip_backends) list(DBI::dbConnect(duckdb::duckdb()))

# Call the testing suite
test_diseasystore(
  diseasystore_generator = DiseasystoreSimulist,
  conn_generator = conns,
  target_schema = "test",
  test_start_date = as.Date("2020-02-20"),
  skip_backends = "SQLiteConnection"
)
```

---

to_diseasystore_case *Transform case definition to PascalCase*

Description

Transform case definition to PascalCase

Usage

```
to_diseasystore_case(label)
```

Arguments

label (character)
: A character string that controls which feature store to get data from.

Value

The given label formatted to match a Diseasystore

Examples

```
to_diseasystore_case("Google COVID-19")  # DiseasystoreGoogleCovid19
```

`%.%` *Existence aware pick operator*

Description

Existence aware pick operator

Usage

```
env %.% field
```

Arguments

`env`	(`object`) The object or environment to attempt to pick from
`field`	(`character`) The name of the field to pick from `env`

Value

Error if the `field` does not exist in `env`, otherwise it returns `field`

Examples

```
t <- list(a = 1, b = 2)

t$a       # 1
t %.% a   # 1

t$c # NULL
try(t %.% c) # Gives error since "c" does not exist in "t"
```

Index

%.%, 21

add_years, 2
age_labels, 3
age_on_date, 4
aggregators, 4, *15*
aggregators(), *16*
available_diseasystores, 6

diseasyoption, 6
diseasystore_exists, 14
DiseasystoreBase, 7, *11–13*
DiseasystoreEcdcRespiratoryViruses, 11
DiseasystoreGoogleCovid19, 12
DiseasystoreSimulist, 13
drop_diseasystore, 14

FeatureHandler, 15

get_diseasystore, 17

key_join_count *(*aggregators*)*, 4
key_join_max *(*aggregators*)*, 4
key_join_min *(*aggregators*)*, 4
key_join_sum *(*aggregators*)*, 4

R6, *7–9*, *11–13*, *15, 16*

source_conn_github
        *(*source_conn_helpers*)*, 18
source_conn_helpers, 18
source_conn_path *(*source_conn_helpers*)*,
        18

test_diseasystore, 19
to_diseasystore_case, 20