Package 'featdelta'

May 13, 2026

Type Package

Title Incremental Feature Engineering with Database Persistence

Version 0.1.0

Author Rudolfs Kregers [aut, cre]

Maintainer Rudolfs Kregers <rudolfs.kregers@gmail.com>

Description Define feature logic, compute only new or unprocessed rows, and persist the resulting flat feature table in a database. The package provides an explicit incremental pipeline for fetching source rows, computing feature definitions, and writing computed features to a database table.

License GPL-3

BugReports https://github.com/LordRudolf/featdelta/issues

Encoding UTF-8

Imports DBI, rlang

Suggests RSQLite, knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

RoxygenNote 7.3.3

NeedsCompilation no

Repository CRAN

Date/Publication 2026-05-13 08:00:02 UTC

Contents

`fd_block` *Define a multi-column feature block*

Description

Creates a block definition object for use inside `fd_define()`.

Usage

```
fd_block(x, expected_names = NULL, envir = NULL)
```

Arguments

`x`	Block definition input. Must be either a function or a single expression-like input that later evaluates to a `data.frame`.
`expected_names`	Optional character vector of expected output column names. If supplied, names must be non-empty, non-`NA`, and unique. These names are not required for normal operation, but they can later be used by `fd_compute()` for validation and for optional NA-completion of expected-but-missing outputs.
`envir`	Optional environment used when normalizing expression-like inputs. If `NULL`, the calling environment is used.

Details

A block represents one ordered definition step that may produce multiple feature columns when later evaluated by `fd_compute()`. Unlike ordinary named single-column definitions, a block may return a whole `data.frame` whose column names become the produced feature names.

Supported block inputs are:

- an inline expression, including braced expressions such as `{ data.frame(...) }`
- a quosure
- an `expression()` object of length 1
- a quoted call or symbol
- a function, intended to receive the current working data and return a `data.frame`

`fd_block()` does not compute anything by itself. It only stores the block definition in a normalized form so that `fd_define()` can include it in the ordered definitions object.

Value

An object of class "featdelta_block".

See Also

Other featdelta defs helpers: fd_define(), print.featdelta_defs(), summary.featdelta_defs()

Examples

```
# Inline expression block
blk <- fd_block({
  data.frame(
    hp_per_cyl = hp / cyl,
    disp_per_cyl = disp / cyl
  )
})

# Function-based block
make_engine_features <- function(data) {
  data.frame(
    hp_per_cyl = data$hp / data$cyl,
    disp_per_cyl = data$disp / data$cyl
  )
}

blk <- fd_block(make_engine_features)

# Optional expected names
blk <- fd_block(
  {
    data.frame(
      hp_per_cyl = hp / cyl
    )
  },
  expected_names = c("hp_per_cyl", "disp_per_cyl")
)
```

---

fd_compute *Compute features from featdelta definitions on in-memory data*

Description

Evaluates a featdelta_defs object created by fd_define() against an in-memory data frame/tibble and returns a feature data frame containing the key column plus computed feature columns.

Usage

```
fd_compute(
  data,
  defs,
  key,
  compute_envir = NULL,
  compute_strict = TRUE,
  verbose = FALSE,
  return_report = FALSE
)
```

Arguments

data	A data frame/tibble containing the raw input variables used to compute features.
defs	A featdelta_defs object created by fd_define().
key	Character scalar naming the primary key column in data.
compute_envir	Optional environment. If supplied, it overrides the stored evaluation environment for expression-based steps.
compute_strict	Logical. If TRUE, any failed step causes the overall compute call to error after the per-step report is assembled. If FALSE, failed column steps return NA columns, and failed block steps return NA columns only when output_names are known in advance.
verbose	Logical. If TRUE, print progress messages and a short summary.
return_report	Logical. If TRUE, return a list with elements data and report. If FALSE, return only the computed data frame and attach the report as attr(x, "fd_report").

Details

fd_compute() is the package's pure in-memory execution step. It does not access the database and does not use the featdelta context system.

Definition steps are evaluated sequentially in the order stored in defs. This means later steps may depend on columns created by earlier steps in the same compute call.

Guardrails implemented by fd_compute() are compute-specific:

- validate the structure of the supplied defs object
- validate result type and row alignment for each step
- preserve 1:1 row alignment with the input data
- reject output-name collisions with key and previously produced names
- attach a per-step computation report

Guardrails that belong at definition time, such as malformed construction of individual definition steps, should be handled by fd_define().

Supported definition step types are:

- ordinary single-column steps

- `fd_block()` multi-column steps

For single-column steps, the result must be a supported vector-like output.

For block steps:

- the result must be a `data.frame`
- the number of rows must match `nrow(data)`
- returned column names must be non-empty and unique
- returned names must not collide with `key`
- returned names must not collide with feature names already produced earlier in the same compute run

If a block declares `expected_names`, then:

- returned names must be a subset of `expected_names`
- missing expected names are added as `NA`
- final output order follows `expected_names`

Because definitions are evaluated sequentially, later steps may use:

- raw input columns from `data`
- columns produced by earlier single-column steps
- columns produced by earlier block steps

Reordering or removing earlier steps may therefore break downstream definitions. When evaluation fails with "object not found"-style errors, `fd_compute()` augments the message with likely reasons.

Value

Either:

- a data frame with class `"featdelta_features"` and attached report attribute, or
- a list `{data, report}` with class `"featdelta_compute_result"` when `return_report = TRUE`

The output preserves row order and row count relative to `data`.

Examples

```
raw_df <- mtcars
raw_df$car_id <- seq_len(nrow(raw_df))

defs <- fd_define(
  hp_per_cyl = hp / cyl,
  engine_ratios = fd_block({
    data.frame(
      disp_per_cyl = disp / cyl,
      wt_per_hp = wt / hp
    )
  }),
```

```
  double_ratio = disp_per_cyl * 2
)

out <- fd_compute(
  data = raw_df,
  defs = defs,
  key = "car_id",
  compute_strict = TRUE
)

head(out)
```

`fd_connect` *Connect to a database with featdelta defaults*

Description

Creates a `featdelta_con` context object from a DBI driver and connection arguments. The returned object stores the live DBI connection plus optional featdelta defaults such as the raw table, feature table, key column, and metadata settings.

Usage

```
fd_connect(
  driver,
  ...,
  raw_table = NULL,
  raw_table_name = NULL,
  feat_table_name = NULL,
  key = NULL,
  meta_enabled = FALSE,
  meta_schema = NULL
)
```

Arguments

`driver`	A DBI driver object, such as `RSQLite::SQLite()`, `RPostgres::Postgres()`, or `RMariaDB::MariaDB()`.
`...`	Additional arguments passed to `DBI::dbConnect()`.
`raw_table`	Optional character scalar naming the raw/source table.
`raw_table_name`	Deprecated alias for `raw_table`.
`feat_table_name`	Optional character scalar naming the feature table.
`key`	Optional character scalar naming the primary key column.
`meta_enabled`	Logical. Whether featdelta metadata tracking is enabled.
`meta_schema`	Optional list of metadata schema settings.

Value

A `featdelta_con` object.

`fd_define` *Define featdelta feature definitions*

Description

Creates a `featdelta_defs` object that stores ordered definition steps in a normalized internal representation suitable for later evaluation by `fd_compute()`.

Usage

```
fd_define(
  ...,
  defs = NULL,
  description = NULL,
  overwrite = FALSE,
  envir = NULL
)
```

Arguments

`...`	Definitions supplied inline. Ordinary named inputs such as `new_feature = hp / cyl` are stored as single-column definition steps.
`defs`	Optional list of programmatically supplied definitions. Supported inputs are the same as for `....`
`description`	Optional character scalar describing the definitions.
`overwrite`	Logical. If `FALSE`, duplicate step names are rejected. If `TRUE`, later steps replace earlier ones.
`envir`	Optional environment used as the explicit evaluation environment for normalized definitions. If `NULL`, environments are taken from the captured inputs.

Details

`fd_define()` is the definitions-construction step of the package. It does not compute features and does not access the database. Its role is to:

- capture user definitions,
- normalize supported input styles to a canonical internal structure,
- enforce definition-level guardrails such as name validity and duplicate handling,
- store the resulting ordered definition steps for later execution.

Definitions are evaluated later by `fd_compute()` in the order they are stored. This means later definitions may depend on columns created by earlier steps. Reordering or removing earlier steps may therefore break downstream definitions.

Supported definition inputs include:

- inline expressions captured from `...`
- quosures
- `expression()` objects of length 1
- quoted calls or names
- direct scalar constants
- `fd_block()` objects for multi-column feature generation

Ordinary named expressions define one output column per step.

`fd_block()` defines a multi-column step. At compute time, a block is expected to return a `data.frame` with one row per input row. The returned column names become the produced feature names. Optionally, expected output names may be declared in advance via `fd_block(expected_names = ...)`.

Direct atomic constants must be scalar. If vectorized behavior is desired, provide an expression instead of embedding a fixed-length atomic vector in the definitions.

Duplicate step handling is part of definitions construction:

- with `overwrite = FALSE`, duplicate step names error;
- with `overwrite = TRUE`, the last definition wins.

Value

A list of class `"featdelta_defs"` containing:

- `steps`: ordered normalized definition steps
- `description`: optional description
- `created_at`: creation time
- `defs_version`: internal version marker
- `envir_policy`: whether environments were inherited or explicit
- `envir`: the explicit environment, if supplied

See Also

Other featdelta defs helpers: `fd_block()`, `print.featdelta_defs()`, `summary.featdelta_defs()`

Examples

```
# Single-column definitions
defs <- fd_define(
  hp_per_cyl = hp / cyl,
  wt_per_hp = wt / hp
)
```

```
# Definitions can also be supplied programmatically
predef <- expression(log(hp))
defs <- fd_define(
  log_hp = predef
)

# Multi-column block using an inline expression
defs <- fd_define(
  engine_ratios = fd_block({
    data.frame(
      hp_per_cyl = hp / cyl,
      disp_per_cyl = disp / cyl
    )
  })
)

# Multi-column block with declared expected names
defs <- fd_define(
  engine_ratios = fd_block(
    {
      data.frame(
        hp_per_cyl = hp / cyl,
        disp_per_cyl = disp / cyl
      )
    },
    expected_names = c("hp_per_cyl", "disp_per_cyl")
  )
)

# Function-based block
make_engine_features <- function(data) {
  data.frame(
    hp_per_cyl = data$hp / data$cyl,
    disp_per_cyl = data$disp / data$cyl
  )
}

defs <- fd_define(
  engine_ratios = fd_block(make_engine_features)
)

# A block can contain a small feature script with temporary variables
defs <- fd_define(
  engine_script = fd_block({
    hp_per_cyl <- hp / cyl
    disp_per_cyl <- disp / cyl

    data.frame(
      hp_per_cyl = hp_per_cyl,
      disp_per_cyl = disp_per_cyl,
      engine_index = hp_per_cyl + disp_per_cyl
    )
```

```
  })
)

# A function-based block can create a variable number of columns in a loop
make_scaled_features <- function(data) {
  vars <- c("hp", "disp", "wt")
  out <- list()

  for (var in vars) {
    center <- mean(data[[var]], na.rm = TRUE)
    spread <- stats::sd(data[[var]], na.rm = TRUE)
    out[[paste0(var, "_scaled")]] <- (data[[var]] - center) / spread
  }

  as.data.frame(out)
}

defs <- fd_define(
  scaled_inputs = fd_block(make_scaled_features)
)
```

`fd_fetch` *Fetch source rows that are not yet present in the features table*

Description

`fd_fetch()` executes a user-supplied SQL `SELECT` query against a database and returns **only** those rows whose primary key (`key`) is **not present** in an existing features table (`feat_table_name`).

Usage

```
fd_fetch(con, sql, key, feat_table_name, use_max_key = FALSE, verbose = FALSE)
```

Arguments

`con`	A live `DBIConnection`, created with `DBI::dbConnect()`.
`sql`	A single SQL string (typically a `SELECT`) that defines the **source dataset** you want to process into features. In other words, it should return the rows you would normally feed into your feature creation pipeline. The query may join multiple tables and apply filters. The query must be usable as a derived table: `FROM (<sql>) AS r`. Trailing semicolons are tolerated and removed.
`key`	Name of the primary key column (character scalar). Must exist in both the result of `sql` and in `feat_table_name`. This is the identifier used to decide whether a source row has already been processed into the features table.
`feat_table_name`	Name of the existing features table in the database (character scalar). Must exist. This table is used only to identify which `key` values are already present.

use_max_key — Logical. If TRUE, fd_fetch() may reduce join work by first computing MAX(feat_table_name.key) and selecting rows with key > max_key (guaranteed not in feat_table_name). If FALSE, fd_fetch() selects all the rows with any key no in feat_table_name.key which may be computationally more expensive.

verbose — Logical. If TRUE, prints the executed SQL.

Details

This function is intentionally not a general-purpose query runner. It always applies a "not yet processed" filter against feat_table_name and errors if that cannot be done honestly (e.g., missing tables/columns).

Important assumption:

fd_fetch() assumes that:

- the sql query is executed on the **same database** (same con) where feat_table_name is stored, and
- the returned key values are comparable to feat_table_name.key.

Cross-database fetching (e.g., pulling data from one database and comparing to a features table stored in another database) is not supported, because the "not present in feat_table_name" filter must be evaluated by the database engine in a single query context.

Value

A data.frame containing only rows returned by sql whose key is not present in feat_table_name. The result includes an attribute attr(x, "fd_fetch") (a list) with metadata such as key, feat_table_name, use_max_key, max_key (if computed), executed_sql, and n_rows.

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE)) {
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
on.exit(DBI::dbDisconnect(con), add = TRUE)

DBI::dbExecute(con, "CREATE TABLE raw (id INTEGER, x INTEGER)")
DBI::dbExecute(con, "CREATE TABLE r_variables_table (id INTEGER)")
DBI::dbExecute(con, "INSERT INTO raw (id, x) VALUES (1,10), (2,20), (3,30), (4,40), (5,50)")
DBI::dbExecute(con, "INSERT INTO r_variables_table (id) VALUES (1), (2), (4)")

# Returns ids 3 and 5 only (rows not yet present in the features table)
new_rows <- fd_fetch(
  con = con,
  sql = "SELECT * FROM raw",
  key = "id",
  feat_table_name = "r_variables_table"
)
}
```

fd_run *Run the featdelta incremental feature pipeline*

Description

Executes the standard featdelta pipeline:

Usage

```
fd_run(
  con,
  sql,
  defs,
  key,
  feat_table_name,
  verbose = FALSE,
  fetch_mode = c("new_only", "all"),
  use_max_key = FALSE,
  fetch_limit = NULL,
  compute_strict = TRUE,
  compute_envir = NULL,
  create_table = "auto",
  alter_table = TRUE,
  update_table = TRUE,
  dialect = NULL,
  chunk_size = NULL,
  fail_fast = TRUE,
  return_data = c("none", "features", "raw", "both"),
  preview_n = 10L,
  ...
)
```

Arguments

con	A live DBI connection.
sql	SQL query used to fetch raw rows.
defs	A featdelta_defs object created by fd_define().
key	Character scalar naming the key column shared by raw data and feature table.
feat_table_name	Character scalar naming the target feature table.
verbose	Logical. If TRUE, print progress messages.
fetch_mode	Fetch mode. "new_only" fetches rows whose key is not yet present in feat_table_name. "all" fetches all rows returned by sql, allowing explicit refresh/backfill runs.
use_max_key	Logical. Passed to fd_fetch() when fetch_mode = "new_only" and the feature table already exists.

fetch_limit	Optional positive row limit applied after fetching. This is intended for previews and small dry development runs, not as a SQL optimizer.
compute_strict	Logical strictness flag passed to fd_compute().
compute_envir	Optional environment override passed to fd_compute().
create_table	Logical or "auto". Passed to fd_upsert().
alter_table	Logical. Passed to fd_upsert().
update_table	Logical. Passed to fd_upsert().
dialect	Optional dialect override. Supported values are "postgres", "sqlite", and "mysql".
chunk_size	Optional chunk size. Passed to fd_upsert().
fail_fast	Logical. If TRUE, stage errors are raised immediately. If FALSE, errors from fetch, compute, or upsert are captured in the returned fd_run_report.
return_data	Controls whether raw and/or computed data should be included in the returned run report.
preview_n	Non-negative number of rows to include in report previews.
...	Reserved for future context options passed to resolve_ctx().

Details

fd_fetch() -> fd_compute() -> fd_upsert()

fd_run() is the orchestration entry point that connects database fetching, in-memory computation, and database upsert into a single incremental run.

fetch_mode = "new_only" is the default incremental mode. If the feature table already exists, fd_fetch() returns only rows whose key is missing from the feature table. If the feature table does not exist yet, all rows returned by sql are fetched because no rows have been processed. This mode is key-based: it does not recompute existing feature-table rows just because feature definitions changed or new feature definitions were added.

fetch_mode = "all" is an explicit refresh/backfill mode. It recomputes all rows returned by sql and passes them to fd_upsert(). With the default update_table = TRUE, existing keys are updated and new keys are inserted. Use this mode when existing feature values should be refreshed, for example after changing a definition or adding a feature that should be backfilled for already-processed keys.

fd_run() does not currently coordinate concurrent writers. For the MVP, avoid running multiple fd_run()/fd_upsert() calls against the same feature table at the same time. Concurrent writes to different feature tables are independent.

The returned upsert report can include extra_columns: columns that exist in the target feature table but are not produced by the current definitions. These columns are left untouched; the package does not drop, rename, or retire columns automatically.

Value

An object of class "fd_run_report" containing stage summaries, timings, row counts, the compute report, and the upsert report. Depending on return_data, it may also contain raw and/or computed feature data.

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE)) {
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
on.exit(DBI::dbDisconnect(con), add = TRUE)

raw_cars <- mtcars
raw_cars$id <- seq_len(nrow(raw_cars))
DBI::dbWriteTable(con, "raw_cars", raw_cars)

defs <- fd_define(
  hp_per_cyl = hp / cyl,
  engine_ratios = fd_block({
    data.frame(
      disp_per_cyl = disp / cyl,
      wt_per_hp = wt / hp
    )
  })
)

res <- fd_run(
  con = con,
  sql = "SELECT * FROM raw_cars",
  defs = defs,
  key = "id",
  feat_table_name = "features",
  verbose = FALSE
)
res$success
}
```

`fd_upsert` *Upsert computed features into a database features table*

Description

Takes a feature `data.frame` produced by `fd_compute()` (one row per entity id) and writes it into a database table, using explicit incremental semantics: new keys are inserted; existing keys are optionally updated.

Usage

```
fd_upsert(
  con,
  features_df,
  feat_table_name,
  key,
  create_table = "auto",
  alter_table = TRUE,
```

```
  update_table = TRUE,
  chunk_size = NULL,
  verbose = TRUE,
  return_report = TRUE,
  dialect = NULL
)
```

Arguments

con	A DBIConnection. Must be a live connection to the database where feat_table_name exists (or will be created).
features_df	A data.frame containing at minimum the primary key column specified by key, plus one or more feature columns to be stored. Requirements: • key must exist in names(features_df). • features_df[[key]] must contain no NA and no duplicate values. • Each feature column should be an atomic vector (numeric, integer, logical, character, Date/POSIXct) that can be written by DBI for the chosen dialect.
feat_table_name	A single string. Name of the database table where features are stored (the "target" table). For dialects supporting schemas (e.g. Postgres), this may be schema-qualified (e.g. "public.my_features"), subject to dialect rules.
key	A single string. Name of the primary key column in both features_df and feat_table_name. This column uniquely identifies each entity/row.
create_table	Logical, or character value auto. If TRUE, create feat_table_name when it does not exist. The created table includes all columns present in features_df and defines PRIMARY KEY(key). If FALSE and feat_table_name is missing, the function errors.If auto, create the table it hasn't been already created.
alter_table	Logical. If TRUE, add missing feature columns found in features_df but not present in feat_table_name using ALTER TABLE ... ADD COLUMN .... Only additive schema changes are performed. If FALSE, the function errors when features_df contains columns not present in feat_table_name.
update_table	Logical. Controls incremental behavior for keys that already exist in feat_table_name: • If TRUE (default), existing keys are updated (upsert). • If FALSE, only new keys are inserted; any conflict with existing keys results in an error (insert-only mode).
chunk_size	Optional integer batch size. If provided, features_df is written in batches of up to chunk_size rows, each merged via a staging table. Use this to limit memory/packet sizes for large writes. If NULL, write in one batch.
verbose	Logical. If TRUE, emit progress messages (e.g., chunk progress, table creation/alteration actions). Does not affect returned results.
return_report	Logical. If TRUE, return a structured fd_upsert_report; if FALSE, return TRUE invisibly after successful side effects.
dialect	Optional dialect override. Supported values are "postgres", "sqlite", and "mysql".

Details

The function is set-based (staging + merge) and avoids "read all ids into R" patterns. Writes are performed in chunks (optional) inside a transaction.

Schema evolution is supported in a restricted, safe form: when `alter_table = TRUE`, missing feature columns are added to the target table via `ALTER TABLE ... ADD COLUMN ...`. No column drops/renames/type changes are performed. Columns that exist in the target table but are not present in `features_df` are left untouched and reported as `extra_columns`.

Incremental semantics

- Insert new keys (keys in `features_df` not present in `feat_table_name`).
- Update existing keys only when `update_table = TRUE`.
- When `update_table = FALSE`, any overlap between staged keys and existing keys is treated as a conflict and aborts the write.

Counts The returned report contains `would_insert` / `would_update` counts, computed as existence-based counts *prior to the merge* (within the transaction). These are not "rows whose values changed", only "rows targeted as insert/update".

Transactions The operation runs in a transaction. If any step fails (schema change, staging write, merge), the transaction is rolled back and the target table is unchanged. Existing target tables used with `update_table = TRUE` must have a primary key or unique constraint/index on `key`. Tables created by `fd_upsert()` get this primary key automatically.

Concurrency `fd_upsert()` does not currently take an explicit table-level lock. Avoid running concurrent writes to the same feature table. Concurrent writes to different feature tables are independent.

Value

An `fd_upsert_report` object (S3) with a structured summary of actions performed:

- `table_created` (logical)
- `columns_added` (character vector)
- `extra_columns` (character vector): target-table columns not present in the incoming features data, excluding `key`
- `counts$would_insert` (integer)
- `counts$would_update` (integer; `0` when `update_table = FALSE`)
- optional per-chunk breakdown (if chunking is used)

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE)) {
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
on.exit(DBI::dbDisconnect(con), add = TRUE)

feats <- data.frame(
  id = c(1, 2, 3),
  f_age = c(10, 20, 30),
  f_flag = c(TRUE, FALSE, TRUE)
```

```
)

# Create table and insert initial rows
r1 <- fd_upsert(
  con = con,
  features_df = feats,
  feat_table_name = "features_tbl",
  key = "id",
  create_table = TRUE,
  alter_table = FALSE,
  update_table = TRUE,
  verbose = FALSE
)

# Upsert: update ids 2-3, insert id 4
feats2 <- data.frame(
  id = c(2, 3, 4),
  f_age = c(21, 31, 40),
  f_flag = c(FALSE, TRUE, FALSE)
)

r2 <- fd_upsert(
  con = con,
  features_df = feats2,
  feat_table_name = "features_tbl",
  key = "id",
  update_table = TRUE,
  verbose = FALSE
)

# Schema evolution: add a new feature column
feats3 <- data.frame(
  id = c(4, 5),
  f_age = c(41, 50),
  f_flag = c(TRUE, TRUE),
  f_new = c("A", "B")
)

r3 <- fd_upsert(
  con = con,
  features_df = feats3,
  feat_table_name = "features_tbl",
  key = "id",
  alter_table = TRUE,
  update_table = TRUE,
  verbose = FALSE
)
}
```

print.fd_run_report *Print a featdelta run report*

Description

Print a featdelta run report

Usage

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

Arguments

x	An fd_run_report object.
...	Unused.

Value

The input object invisibly.

print.featdelta_con *Print a featdelta connection context*

Description

Print a featdelta connection context

Usage

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

Arguments

x	A featdelta_con object.
...	Unused.

Value

The input object invisibly.

print.featdelta_defs *Print a featdelta definitions object*

Description

Prints a concise human-readable summary of a `featdelta_defs` object.

Usage

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

Arguments

`x`	A `featdelta_defs` object.
`...`	Unused.

Value

The input object invisibly.

See Also

Other featdelta defs helpers: `fd_block()`, `fd_define()`, `summary.featdelta_defs()`

summary.featdelta_defs *Summarize a featdelta definitions object*

Description

Returns a compact data-frame summary of the ordered definition steps stored in a `featdelta_defs` object.

Usage

```
## S3 method for class 'featdelta_defs'
summary(object, ...)
```

Arguments

`object`	A `featdelta_defs` object.
`...`	Unused.

Value

A data frame with one row per definition step and columns such as step name, type, mode, declared outputs, and stored expression text.

See Also

Other featdelta defs helpers: `fd_block()`, `fd_define()`, `print.featdelta_defs()`

Index