basetable: A Complete Function Reference

Every exported function, explained progressively with real examples

Contents

1 Introduction **3**

2 Core verbs **3**
- 2.1 Subsetting rows and columns: `subset()` . . . 4
- 2.2 Picking and dropping columns: `pick()`, `drop()` . . . 4
- 2.3 Creating and modifying columns: `transform()`, `within()` . . . 5
- 2.4 Renaming columns: `renamecols()` . . . 6

3 Aggregation and grouping **7**
- 3.1 Grouped summaries: `aggregate()` . . . 7
- 3.2 Counting rows: `count()`, `propcount()` . . . 7
- 3.3 Named summaries with `summaries()` . . . 7
- 3.4 Splitting into groups: `split()` and `applyby()` . . . 7
- 3.5 First/last row per group: `firstby()`, `lastby()` . . . 8

4 Joining tables **8**
- 4.1 Standard joins: `merge()` . . . 8
- 4.2 Semi- and anti-joins: `semimerge()`, `antimerge()` . . . 9
- 4.3 Key-matching helpers: `matchedkeys()`, `unmatchedkeys()`, `joinrelationship()` . . . 9
- 4.4 Cross joins: `crossmerge()` . . . 9
- 4.5 Rolling and nearest-key joins: `rollingmerge()`, `nearestmerge()` . . . 10
- 4.6 Interval overlap joins: `overlapmerge()` . . . 10
- 4.7 Updating values from another table: `updatemerge()` . . . 10
- 4.8 Conditional joins: `nonequimerge()`, `rangemerge()` . . . 11

5 Set operations and table comparison **11**
- 5.1 Row-level set operations: `unionrows()`, `intersectrows()`, `diffrows()` . . . 11
- 5.2 Comparing two tables: `equalrows()`, `equaldata()`, `sameschema()`, `compareschema()`, `changedcols()` . . . 12
- 5.3 Comparing two snapshots of the same table: `addedrows()`, `removedrows()`, `changedrows()` 12
- 5.4 Row/column binding: `rbindfill()`, `cbind()` . . . 13

6 Reshaping **13**
- 6.1 Long and wide formats: `tolong()`, `towide()` . . . 13
- 6.2 Base R reshape, stack, and unstack: `reshape()`, `stack()`, `unstack()` . . . 14
- 6.3 Splitting and combining text columns: `separate()`, `unite()` . . . 14
- 6.4 Transposing: `transpose()` . . . 14

7 Exploration and EDA **15**
- 7.1 Shape and types: `dims()`, `types()`, `nrows()`, `ncols()` . . . 15
- 7.2 Peeking at data: `headtail()`, `preview()` . . . 15
- 7.3 Descriptive statistics: `describe()`, `profile()` . . . 15
- 7.4 Frequency tables: `freq()` . . . 16

7.5 Table 1-style summaries: `summarytab()` . . . 16
7.6 Missingness: `missingness()` . . . 16
7.7 Comparing two tables at a glance: `compare()` . . . 16

8 Data cleaning and validation **17**
8.1 Assertions that stop on failure: `assert*()` . . . 17
8.2 Finding rows/values that fail a check: `invalidrows()`, `invalidvalues()`, `outofrange()` . . . 18
8.3 Missing-data helpers: `missingrows()`, `keepmissing()`, `omitmissing()`, `missingindicator()` 18
8.4 Filling in missing combinations: `completegrid()`, `expandrows()` . . . 19
8.5 Recoding values: `naif()`, `nato()`, `blanktona()`, `natoblank()`, `replacevalues()`, `replacewhere()`, `replacecols()` . . . 19
8.6 Deduplication: `uniquerows()`, `removeduplicates()`, `duplicaterows()`, `duplicatekeys()`, `duplicatenames()` . . . 20
8.7 Column-name hygiene: `cleannames()`, `repairnames()`, `renamewith()`, `commonnames()` . . . 21
8.8 Filling in missing values within groups: `filldown()`, `fillup()`, `fillboth()` . . . 21

9 Row and column utilities **22**
9.1 Column metadata: `colnames()`, `rownames()`, `classes()`, `uniques()`, `cardinality()`, `constants()`, `emptycols()` . . . 22
9.2 Finding blank rows: `emptyrows()` . . . 22
9.3 Reordering columns: `move()`, `firstcols()`, `lastcols()` . . . 22
9.4 Row subsets: `firstrows()`, `lastrows()`, `samplerows()`, `samplefrac()`, `reverse()` . . . 23
9.5 Sorting rows: `orderrows()` . . . 23
9.6 Row position helpers: `rownumber()` . . . 24
9.7 Row-wise reductions across columns: `rowmin()`, `rowmax()`, `rowany()`, `rowall()`, `rowcount()`, `rowfirst()`, `rowlast()`, `rowapply()` . . . 24
9.8 Applying a function to selected columns: `applycols()`, `convertcols()` . . . 24

10 String manipulation **25**
10.1 Whitespace and case: `trim()`, `squish()`, `lower()`, `upper()`, `titlecase()`, `sentencecase()`, `textlen()` . . . 25
10.2 Substrings and padding: `left()`, `right()`, `middle()`, `truncate()`, `padleft()`, `padright()`, `padcenter()` . . . 25
10.3 Pattern matching: `containstext()`, `matchestext()`, `startswith()`, `endswith()`, `countmatch()`, `locate()`, `locateall()` . . . 26
10.4 Extraction: `extract()`, `extractall()`, `extractnum()`, `extractint()`, `extractbetween()` . . . 26
10.5 Replace and remove: `replacetext()`, `removetext()`, `removeall()`, `replaceall()` . . . 26
10.6 Split and join: `splittext()`, `splitfirst()`, `splitlast()`, `jointext()`, `collapsetext()` . . . 27
10.7 Blank and encoding helpers: `isblank()`, `removeaccents()`, `normalizeunicode()`, `normalizeencoding()`, `transliterate()` . . . 27
10.8 String distance and similarity: `textdist()`, `nearesttext()`, `similartext()` . . . 27
10.9 Classification predicates: `isalpha()`, `isalphanumeric()`, `isnumerictext()`, `isintegertext()`, `isemail()`, `isurl()` . . . 27
10.10 Recoding and grouping values: `recode()`, `collapsevalues()`, `casewhen()`, `collapselevels()`, `lump()`, `reorderlevels()`, `expandlevels()` . . . 28

11 Parsing text into typed values **29**

12 Dates and times **29**
12.1 Extracting components . . . 29
12.2 Arithmetic and sequences: `adddays()`, `addweeks()`, `addmonths()`, `addyears()`, `datediff()`, `dateseq()`, `betweendates()` . . . 30
12.3 Rounding dates: `floordate()`, `ceilingdate()`, `rounddate()` . . . 30

13 Numeric transforms and rolling windows **30**

13.1 Rescaling and standardizing: `rescale()`, `standardize()`, `center()`, `winsorize()` 30
13.2 Binning: `quantilegroup()` 31
13.3 Period-over-period change and rank: `percentchange()`, `percentrank()`, `denserank()` 31
13.4 Cumulative helpers: `cumcount()`, `cumedist()`, `cumavg()` 31
13.5 Lag/lead and differencing: `difference()`, `lagvalue()`, `leadvalue()` 31
13.6 Rolling window functions: `rollmean()`, `rollsum()`, `rollmin()`, `rollmax()`, `rollmedian()`, `rollsd()`, `rollvar()`, `rollprod()`, `rollapply()` 32

14 Functional helpers **32**

15 Closing notes **33**

1 Introduction

`basetable` is for people who already know base R and want fast table operations without learning `data.table`'s `[i, j, by]` syntax or dplyr's tidy evaluation. It is aimed specifically at two audiences: **teaching** table manipulation to people learning base R (without the extra cognitive load of non-standard evaluation or terse `[i, j, by]` syntax), and **migrating** a codebase already built on base-R table semantics (`subset()`, `merge()`, `aggregate()`, and friends) onto a faster engine without a rewrite. It is not trying to replace dplyr or data.table for a new project that's free to pick any tool; both are more established choices with a larger ecosystem.

Design goals:

- **Base-style naming and semantics.** Functions read like `subset()`, `transform()`, `aggregate()`, `merge()`, and `split()`, the base R verbs most R users already know, rather than inventing a new grammar.
- **A native C++ execution engine.** Every operation that touches real data routes through basetable's own compiled kernels, so the package is fast without asking you to learn `data.table`'s `[i, j, by]` syntax and without a `data.table` dependency.
- **Explicit, standard-evaluation interfaces.** Column names are passed as character strings, not bare symbols captured by non-standard evaluation (with a few clearly-marked exceptions, like `subset()`'s and `transform()`'s expression arguments, which mirror base R's own `subset()`/`transform()` behavior).

`basetable` deliberately does **not** ship the dplyr-coined verbs (`filter()`, `select()`, `mutate()`, `arrange()`, `summarise()`, `distinct()`, `glimpse()`, `slice()`, `relocate()`, `bind_rows()`, `bind_cols()`), so it can be attached alongside `dplyr` without shadowing its grammar. The two names it does share with `dplyr` are `count()` and `pick()`, kept because they read as base-style verbs. It also reuses true base-R names like `subset()`, `merge()`, `transform()` and `split()`, which `data.table` also defines. Whenever two attached packages export the same name the one attached **last** wins and silently shadows the other: call `basetable::fn()` explicitly (or use the `conflicted` package) if you need both loaded at once.

This document walks through **every function `basetable` exports**, grouped into sections that build on each other: start with the core verbs, move through aggregation, joining, reshaping, and exploration, then into the lower-level data-cleaning, string, date, and numeric utility functions. Read it top to bottom to learn the package, or jump to a section as a reference.

Every example below is real, runnable code: the output shown is the actual output `basetable` produces.

2 Core verbs

These are the everyday functions: subsetting rows, picking columns, and creating new columns. If you know `mtcars`, you know enough to follow along.

2.1 Subsetting rows and columns: `subset()`

`subset()` mirrors `base::subset()` exactly: a logical condition for rows, and an optional `select` for columns.

```
subset(mtcars, cyl == 6, select = c("mpg", "hp", "wt"))
```

```
## # basetable: 7 x 3
##    mpg  hp    wt
## 1 21.0 110 2.620
## 2 21.0 110 2.875
## 3 21.4 110 3.215
## 4 18.1 105 3.460
## 5 19.2 123 3.440
## 6 17.8 123 3.440
## 7 19.7 175 2.770
```

Pass `by` to evaluate the condition **within each group**, so an aggregate inside it is per group. This is basetable's grouped filter; there is no stateful `group_by()`, the grouping is named at the call.

```
subset(mtcars, mpg > mean(mpg), by = "cyl")
```

```
## # basetable: 16 x 11
##     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1  21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## 2  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## 3  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## 4  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## 5  16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## 6  17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## 7  15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## 8  32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## 9  30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## 10 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## # 6 more rows
```

2.2 Picking and dropping columns: `pick()`, `drop()`

`pick()` keeps only the named columns; `drop()` removes them.

```
pick(mtcars, c("mpg", "hp"))
```

```
## # basetable: 32 x 2
##     mpg  hp
## 1  21.0 110
## 2  21.0 110
## 3  22.8  93
## 4  21.4 110
## 5  18.7 175
## 6  18.1 105
## 7  14.3 245
## 8  24.4  62
## 9  22.8  95
## 10 19.2 123
## # 22 more rows
```

```
drop(mtcars, c("vs", "am"))
```

```
## # basetable: 32 x 9
```

```
##     mpg cyl  disp  hp drat    wt  qsec gear carb
## 1  21.0   6 160.0 110 3.90 2.620 16.46    4    4
## 2  21.0   6 160.0 110 3.90 2.875 17.02    4    4
## 3  22.8   4 108.0  93 3.85 2.320 18.61    4    1
## 4  21.4   6 258.0 110 3.08 3.215 19.44    3    1
## 5  18.7   8 360.0 175 3.15 3.440 17.02    3    2
## 6  18.1   6 225.0 105 2.76 3.460 20.22    3    1
## 7  14.3   8 360.0 245 3.21 3.570 15.84    3    4
## 8  24.4   4 146.7  62 3.69 3.190 20.00    4    2
## 9  22.8   4 140.8  95 3.92 3.150 22.90    4    2
## 10 19.2   6 167.6 123 3.92 3.440 18.30    4    4
## # 22 more rows
```

2.3 Creating and modifying columns: `transform()`, `within()`

`transform()` adds or replaces columns using named expressions, exactly like `base::transform()`. Later expressions in the same call can refer to columns created earlier in that call.

```
transform(mtcars, power = hp / wt, power_sq = power^2)
```

```
## # basetable: 32 x 13
##     mpg cyl  disp  hp drat    wt  qsec vs am gear carb power power_sq
## 1  21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4 41.98   1762.7
## 2  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4 38.26   1463.9
## 3  22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1 40.09   1606.9
## 4  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1 34.21   1170.6
## 5  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2 50.87   2588.0
## 6  18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1 30.35    920.9
## 7  14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4 68.63   4709.7
## 8  24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2 19.44    377.7
## 9  22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2 30.16    909.5
## 10 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4 35.76   1278.5
## # 22 more rows
```

`within()` instead evaluates a whole block of code against the data's columns and returns whatever new/changed variables come out: this is `basetable`'s version of `base::within()`.

```
within(mtcars, {
  power <- hp / wt
  heavy <- wt > 3.5
})
```

```
## # basetable: 32 x 13
##    heavy power carb gear am vs  qsec    wt drat  hp  disp cyl  mpg
## 1  FALSE 41.98    4    4  1  0 16.46 2.620 3.90 110 160.0   6 21.0
## 2  FALSE 38.26    4    4  1  0 17.02 2.875 3.90 110 160.0   6 21.0
## 3  FALSE 40.09    1    4  1  1 18.61 2.320 3.85  93 108.0   4 22.8
## 4  FALSE 34.21    1    3  0  1 19.44 3.215 3.08 110 258.0   6 21.4
## 5  FALSE 50.87    2    3  0  0 17.02 3.440 3.15 175 360.0   8 18.7
## 6  FALSE 30.35    1    3  0  1 20.22 3.460 2.76 105 225.0   6 18.1
## 7   TRUE 68.63    4    3  0  0 15.84 3.570 3.21 245 360.0   8 14.3
## 8  FALSE 19.44    2    4  0  1 20.00 3.190 3.69  62 146.7   4 24.4
## 9  FALSE 30.16    2    4  0  1 22.90 3.150 3.92  95 140.8   4 22.8
## 10 FALSE 35.76    4    4  0  1 18.30 3.440 3.92 123 167.6   6 19.2
## # 22 more rows
```

2.4 Renaming columns: `renamecols()`

`renamecols()` renames columns using `new = old` pairs. Old column names may be supplied as bare names or character strings.

```
mtcars |>
  subset(cyl == 6 & mpg > 18, select = c("mpg", "cyl", "hp")) |>
  transform(ratio = hp / mpg) |>
  orderrows(by = "ratio", decreasing = TRUE)
```

```
## # basetable: 6 x 4
##    mpg cyl  hp ratio
## 1 19.7   6 175 8.883
## 2 19.2   6 123 6.406
## 3 18.1   6 105 5.801
## 4 21.0   6 110 5.238
## 5 21.0   6 110 5.238
## 6 21.4   6 110 5.140
```

```
transform(mtcars, ratio = hp / mpg, .keep = FALSE)
```

```
## # basetable: 32 x 1
##     ratio
## 1   5.238
## 2   5.238
## 3   4.079
## 4   5.140
## 5   9.358
## 6   5.801
## 7  17.133
## 8   2.541
## 9   4.167
## 10  6.406
## # 22 more rows
```

```
renamecols(mtcars, horsepower = hp) |> pick(c("horsepower", "mpg"))
```

```
## # basetable: 32 x 2
##    horsepower  mpg
## 1         110 21.0
## 2         110 21.0
## 3          93 22.8
## 4         110 21.4
## 5         175 18.7
## 6         105 18.1
## 7         245 14.3
## 8          62 24.4
## 9          95 22.8
## 10        123 19.2
## # 22 more rows
```

3 Aggregation and grouping

3.1 Grouped summaries: `aggregate()`

`aggregate()` computes one summary function over one or more value columns, per group, a faster analogue of `stats::aggregate()`. Note it returns a `bt_table` directly, since it's meant to be fast and composable with further basetable operations.

```
aggregate(mtcars, by = "cyl", value = c("mpg", "hp"), fun = mean)
```

```
## # basetable: 3 x 3
##   cyl   mpg     hp
## 1   4 26.66  82.64
## 2   6 19.74 122.29
## 3   8 15.10 209.21
```

3.2 Counting rows: `count()`, `propcount()`

`count()` counts rows per group; `propcount()` adds proportions, optionally computed within a `margin` of columns rather than over the whole table.

```
count(iris, by = "Species")
```

```
## # basetable: 3 x 2
##      Species  n
## 1     setosa 50
## 2 versicolor 50
## 3  virginica 50
```

```
propcount(mtcars, by = c("cyl", "am"), margin = "cyl")
```

```
## # basetable: 6 x 4
##   cyl am  n   prop
## 1   6  1  3 0.4286
## 2   4  1  8 0.7273
## 3   6  0  4 0.5714
## 4   8  0 12 0.8571
## 5   4  0  3 0.2727
## 6   8  1  2 0.1429
```

3.3 Named summaries with `summaries()`

`summaries()` takes named summary expressions directly, with `by` as an argument, useful when you're calling it programmatically:

```
summaries(mtcars, "cyl", mean_hp = mean(hp), sd_hp = sd(hp))
```

```
## # basetable: 3 x 3
##   cyl mean_hp sd_hp
## 1   6  122.29 24.26
## 2   4   82.64 20.93
## 3   8  209.21 50.98
```

3.4 Splitting into groups: `split()` and `applyby()`

`split()` breaks a table into a named list of `bt_tables`, one per group, using the native grouping engine internally:

```r
pieces <- split(iris, by = "Species")
names(pieces)
```

```
## [1] "setosa"     "versicolor" "virginica"
```

```r
pieces$setosa |> headtail(2)
```

```
## # basetable: 4 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          5.3         3.7          1.5         0.2  setosa
## 4          5.0         3.3          1.4         0.2  setosa
```

`applyby()` applies a function to each group through the same grouping split path and, if `bind = TRUE`, stitches the results back into one table:

```r
applyby(mtcars, by = "cyl", fun = function(d) data.frame(mean_mpg = mean(d$mpg)), bind = TRUE, id = "cy
```

```
## # basetable: 3 x 2
##   cyl_group mean_mpg
## 1         6    19.74
## 2         4    26.66
## 3         8    15.10
```

3.5 First/last row per group: `firstby()`, `lastby()`

```r
df <- data.frame(id = c(1, 1, 2, 2, 2), visit = c(1, 2, 1, 2, 3), value = c(10, 12, 5, 6, 7))
firstby(df, by = "id")
```

```
## # basetable: 2 x 3
##   id visit value
## 1  1     1    10
## 2  2     1     5
```

```r
lastby(df, by = "id")
```

```
## # basetable: 2 x 3
##   id visit value
## 1  1     2    12
## 2  2     3     7
```

4 Joining tables

`basetable` provides the full family of table joins, all engine-backed.

4.1 Standard joins: `merge()`

`merge()` mirrors `base::merge()`'s interface exactly (`by`, `all`, `all.x`, `all.y`, `suffixes`) while running on the native join engine internally.

```r
orders <- data.frame(id = c(1, 2, 3), customer = c("a", "b", "c"))
payments <- data.frame(id = c(2, 3, 4), amount = c(50, 75, 20))

merge(orders, payments, by = "id")
```

```
## # basetable: 2 x 3
##   id customer amount
## 1  2        b     50
## 2  3        c     75
```

```
merge(orders, payments, by = "id", all.x = TRUE)
```

```
## # basetable: 3 x 3
##   id customer amount
## 1  1        a     NA
## 2  2        b     50
## 3  3        c     75
```

4.2 Semi- and anti-joins: `semimerge()`, `antimerge()`

`semimerge()` keeps rows of `x` whose key exists in `y` (like `merge()` but without adding `y`'s columns); `antimerge()` keeps rows of `x` whose key does **not** exist in `y`.

```
semimerge(orders, payments, by = "id")
```

```
## # basetable: 2 x 2
##   id customer
## 1  2        b
## 2  3        c
```

```
antimerge(orders, payments, by = "id")
```

```
## # basetable: 1 x 2
##   id customer
## 1  1        a
```

4.3 Key-matching helpers: `matchedkeys()`, `unmatchedkeys()`, `joinrelationship()`

`matchedkeys()`/`unmatchedkeys()` return `x`'s distinct rows whose key is (or isn't) present in `y`. `joinrelationship()` classifies the cardinality of a join key between two tables.

```
matchedkeys(orders, payments, by = "id")
```

```
## # basetable: 2 x 2
##   id customer
## 1  2        b
## 2  3        c
```

```
unmatchedkeys(orders, payments, by = "id")
```

```
## # basetable: 1 x 2
##   id customer
## 1  1        a
```

```
joinrelationship(orders, payments, by = "id")
```

```
## [1] "one-to-one"
```

4.4 Cross joins: `crossmerge()`

`crossmerge()` returns the full Cartesian product of two tables.

```
crossmerge(data.frame(size = c("S", "M")), data.frame(color = c("red", "blue")))
```

```
## # basetable: 4 x 2
##   size color
## 1    S   red
## 2    S  blue
## 3    M   red
## 4    M  blue
```

4.5 Rolling and nearest-key joins: `rollingmerge()`, `nearestmerge()`

`rollingmerge()` matches each row of `x` to the nearest row of `y` **at or before** it (`direction = "backward"`), at or after it (`"forward"`), or whichever is numerically closest (`"nearest"`), the classic "as-of" join used for time series and event logs. `nearestmerge()` is a convenience wrapper for `direction = "nearest"`.

```
trades <- data.frame(id = 1L, time = c(5, 10, 15))
quotes <- data.frame(id = 1L, time = c(3, 8, 14), price = c(100, 101, 99))

rollingmerge(trades, quotes, by = c("id", "time"), direction = "backward")
```

```
## # basetable: 3 x 3
##   id time price
## 1  1    5   100
## 2  1   10   101
## 3  1   15    99
```

```
nearestmerge(trades, quotes, by = c("id", "time"))
```

```
## # basetable: 3 x 3
##   id time price
## 1  1    5   100
## 2  1   10   101
## 3  1   15    99
```

4.6 Interval overlap joins: `overlapmerge()`

`overlapmerge()` matches rows whose `[startx, endx]` interval overlaps a `[starty, endy]` interval in `y` (optionally within an exact `by` key), for example, matching events to the time windows they fall inside.

```
events <- data.frame(id = 1L, start = c(1, 10), end = c(5, 15))
windows <- data.frame(id = 1L, start = c(0, 8), end = c(6, 20), label = c("early", "late"))

overlapmerge(events, windows, startx = "start", endx = "end", starty = "start", endy = "end", by = "id"
```

```
## # basetable: 2 x 6
##   id start end start end label
## 1  1     1   5     0   6 early
## 2  1    10  15     8  20  late
```

4.7 Updating values from another table: `updatemerge()`

`updatemerge()` overwrites `x`'s values with `y`'s wherever the join key matches (an in-place "upsert" of column values, not a column-adding merge).

```
current <- data.frame(id = c(1, 2, 3), status = c("pending", "pending", "pending"))
updates <- data.frame(id = c(2, 3), status = c("shipped", "cancelled"))

updatemerge(current, updates, by = "id")
```

```
## # basetable: 3 x 2
##   id    status
## 1  1   pending
## 2  2   shipped
## 3  3 cancelled
```

4.8 Conditional joins: `nonequimerge()`, `rangemerge()`

`nonequimerge()` supports genuine inequality conditions in `by`, written as `"xcol<op>ycol"`, alongside any plain exact-match columns:

```
events <- data.frame(id = 1, date = as.Date("2024-01-15"))
periods <- data.frame(
  id = 1,
  start_date = as.Date(c("2024-01-01", "2024-02-01")),
  end_date = as.Date(c("2024-01-31", "2024-02-28")),
  period = c("Jan", "Feb")
)
nonequimerge(events, periods, by = c("id", "date>=start_date", "date<=end_date"))
```

```
## # basetable: 1 x 5
##   id       date start_date   end_date period
## 1  1 2024-01-15 2024-01-01 2024-01-31    Jan
```

`rangemerge()` keeps every row of `x` (a table of ranges), attaching the row(s) of `y` whose `value` column falls inside that range, matched within `by`. An `x` row with no `y` row in range still appears once, with `NA` for `y`'s columns.

```
ranges <- data.frame(id = c(1, 1, 2), lower = c(0, 0, 0), upper = c(10, 10, 20))
points <- data.frame(id = c(1, 1, 2), val = c(5, 15, 5), label = c("a", "b", "c"))
rangemerge(ranges, points, by = "id", lower = "lower", upper = "upper", value = "val")
```

```
## # basetable: 3 x 5
##   id lower upper val label
## 1  1     0    10   5     a
## 2  1     0    10   5     a
## 3  2     0    20   5     c
```

5 Set operations and table comparison

5.1 Row-level set operations: `unionrows()`, `intersectrows()`, `diffrows()`

```
a <- data.frame(id = c(1, 2, 3))
b <- data.frame(id = c(2, 3, 4))

unionrows(a, b)
```

```
## # basetable: 4 x 1
##   id
## 1  1
## 2  2
## 3  3
## 4  4
```

```
intersectrows(a, b, by = "id")
```

```
## # basetable: 2 x 1
##   id
## 1  2
## 2  3
```

```
diffrows(a, b, by = "id")
```

```
## # basetable: 1 x 1
##   id
## 1  1
```

5.2 Comparing two tables: `equalrows()`, `equaldata()`, `sameschema()`, `compareschema()`, `changedcols()`

`equalrows()` aligns two tables by a key and compares every column (order-insensitive, duplicate-sensitive), not just the key itself. `equaldata()` compares full table content directly. `sameschema()` and `compareschema()` (aliased for this purpose by `changedcols()`) compare column names and types.

```
equalrows(a, data.frame(id = c(3, 2, 1)), by = "id")
```

```
## [1] TRUE
```

```
equaldata(a, a[nrow(a):1, , drop = FALSE], ignoreorder = TRUE)
```

```
## [1] TRUE
```

```
sameschema(mtcars, mtcars)
```

```
## [1] TRUE
```

```
compareschema(mtcars, iris)
```

```
## # basetable: 16 x 5
##    column in_x  in_y type_x type_y
## 1     mpg TRUE FALSE double   <NA>
## 2     cyl TRUE FALSE double   <NA>
## 3    disp TRUE FALSE double   <NA>
## 4      hp TRUE FALSE double   <NA>
## 5    drat TRUE FALSE double   <NA>
## 6      wt TRUE FALSE double   <NA>
## 7    qsec TRUE FALSE double   <NA>
## 8      vs TRUE FALSE double   <NA>
## 9      am TRUE FALSE double   <NA>
## 10   gear TRUE FALSE double   <NA>
## # 6 more rows
```

5.3 Comparing two snapshots of the same table: `addedrows()`, `removedrows()`, `changedrows()`

`changedrows()` returns only the rows whose key exists in both snapshots *and* whose non-key values actually differ: a row whose key matches but whose values are identical is not "changed".

```
old <- data.frame(id = c(1, 2, 3), status = c("a", "b", "c"))
new <- data.frame(id = c(2, 3, 4), status = c("b", "c2", "d"))

addedrows(old, new, by = "id")
```

```
## # basetable: 1 x 2
```

```
##   id status
## 1  4      d
```

```
removedrows(old, new, by = "id")
```

```
## # basetable: 1 x 2
##   id status
## 1  1      a
```

```
changedrows(old, new, by = "id")
```

```
## # basetable: 1 x 3
##   id status.old status.new
## 1  3          c         c2
```

5.4 Row/column binding: `rbindfill()`, `cbind()`

`rbindfill()` stacks tables, filling missing columns with `NA`. For binding tables side by side, use base R's own `cbind()`.

```
rbindfill(data.frame(x = 1), data.frame(x = 2, y = 3))
```

```
## # basetable: 2 x 2
##   x  y
## 1 1 NA
## 2 2  3
```

```
cbind(data.frame(a = 1:2), data.frame(b = 3:4))
```

```
##   a b
## 1 1 3
## 2 2 4
```

6 Reshaping

6.1 Long and wide formats: `tolong()`, `towide()`

```
wide <- data.frame(id = 1:2, jan = c(10, 20), feb = c(11, 22))
long <- tolong(wide, cols = c("jan", "feb"), names = "month", values = "sales")
long
```

```
## # basetable: 4 x 3
##   id month sales
## 1  1   jan    10
## 2  2   jan    20
## 3  1   feb    11
## 4  2   feb    22
```

```
towide(long, names = "month", values = "sales", idcols = "id", fun = sum)
```

```
## # basetable: 2 x 3
##   id jan feb
## 1  1  10  11
## 2  2  20  22
```

6.2 Base R reshape, stack, and unstack: `reshape()`, `stack()`, `unstack()`

`reshape()`, `stack()`, and `unstack()` are `basetable`-native wrappers around `stats::reshape()`, `utils::stack()`, and `utils::unstack()`, kept for base-R compatibility rather than reimplemented on the engine.

```
wide2 <- data.frame(id = 1:2, t1 = c(10, 20), t2 = c(11, 22))
reshape(wide2, varying = c("t1", "t2"), v.names = "value", timevar = "time", idvar = "id", direction =
```

```
##     id time value
## 1.1  1    1    10
## 2.1  2    1    20
## 1.2  1    2    11
## 2.2  2    2    22
```

```
stack(data.frame(a = 1:2, b = 3:4))
```

```
## # basetable: 4 x 2
##   values ind
## 1      1   a
## 2      2   a
## 3      3   b
## 4      4   b
```

6.3 Splitting and combining text columns: `separate()`, `unite()`

```
codes <- data.frame(code = c("A-1-red", "B-2-blue"))
parts <- separate(codes, column = "code", into = c("letter", "num", "color"), sep = "-")
parts
```

```
## # basetable: 2 x 3
##   letter num color
## 1      A   1   red
## 2      B   2  blue
```

```
unite(parts, column = "code", cols = c("letter", "num", "color"), sep = "_")
```

```
## # basetable: 2 x 1
##       code
## 1  A_1_red
## 2 B_2_blue
```

6.4 Transposing: `transpose()`

```
transpose(data.frame(a = 1:2, b = 3:4))
```

```
## # basetable: 2 x 2
##   V1 V2
## 1  1  2
## 2  3  4
```

7 Exploration and EDA

7.1 Shape and types: `dims()`, `types()`, `nrows()`, `ncols()`

```
dims(iris)
```

```
## # basetable: 1 x 2
##   rows cols
## 1  150    5
```

```
types(iris)
```

```
## # basetable: 5 x 3
##         column   class  typeof
## 1 Sepal.Length numeric  double
## 2  Sepal.Width numeric  double
## 3 Petal.Length numeric  double
## 4  Petal.Width numeric  double
## 5      Species  factor integer
```

7.2 Peeking at data: `headtail()`, `preview()`

```
headtail(iris, 2)
```

```
## # basetable: 4 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          5.1         3.5          1.4         0.2    setosa
## 2          4.9         3.0          1.4         0.2    setosa
## 3          6.2         3.4          5.4         2.3 virginica
## 4          5.9         3.0          5.1         1.8 virginica
```

```
preview(iris)
```

```
## Rows: 150  Columns: 5
## $ Sepal.Length <numeric> 5.1, 4.9, 4.7
## $ Sepal.Width <numeric> 3.5, 3, 3.2
## $ Petal.Length <numeric> 1.4, 1.4, 1.3
## $ Petal.Width <numeric> 0.2, 0.2, 0.2
## $ Species <factor> setosa, setosa, setosa
```

7.3 Descriptive statistics: `describe()`, `profile()`

`describe()` gives a per-column summary (mean, sd, quantiles for numeric columns; top values for others). `profile()` is an alias.

```
describe(iris)
```

```
## # basetable: 5 x 14
##         column   class   n missing missing_prop distinct  mean     sd min q25 median q75
## 1 Sepal.Length numeric 150       0            0       35 5.843 0.8281 4.3 5.1   5.80 6.4
## 2  Sepal.Width numeric 150       0            0       23 3.057 0.4359 2.0 2.8   3.00 3.3
## 3 Petal.Length numeric 150       0            0       43 3.758 1.7653 1.0 1.6   4.35 5.1
## 4  Petal.Width numeric 150       0            0       22 1.199 0.7622 0.1 0.3   1.30 1.8
## 5      Species  factor 150       0            0        3    NA     NA  NA  NA     NA  NA
##   max                                      top
## 1 7.9
## 2 4.4
```

```
## 3 6.9
## 4 2.5
## 5  NA setosa (50), versicolor (50), virginica (50)
```

7.4 Frequency tables: `freq()`

```
freq(iris, column = "Species")
```

```
## # basetable: 3 x 2
##      Species  n
## 1     setosa 50
## 2 versicolor 50
## 3  virginica 50
```

```
freq(mtcars, column = "cyl", by = "am", prop = TRUE)
```

```
## # basetable: 6 x 4
##   am cyl  n   prop
## 1  0   8 12 0.6316
## 2  1   4  8 0.6154
## 3  0   6  4 0.2105
## 4  1   6  3 0.2308
## 5  0   4  3 0.1579
## 6  1   8  2 0.1538
```

7.5 Table 1-style summaries: `summarytab()`

```
dat <- transform(mtcars, am = factor(am, labels = c("Automatic", "Manual")))
summarytab(dat, vars = c("mpg", "cyl"), by = "am", p_value = TRUE)
```

```
## # basetable: 2 x 6
##   variable     level  Automatic     Manual    Overall p_value
## 1      mpg Mean (SD) 17.1 (3.8) 24.4 (6.2) 20.1 (6.0) 0.00137
## 2      cyl Mean (SD)  6.9 (1.5)  5.1 (1.6)  6.2 (1.8) 0.00246
```

7.6 Missingness: `missingness()`

```
with_na <- transform(iris, Sepal.Length = ifelse(Sepal.Length > 7, NA, Sepal.Length))
missingness(with_na)
```

```
## # basetable: 5 x 4
##         column missing missing_prop complete
## 1 Sepal.Length      12         0.08      138
## 2  Sepal.Width       0         0.00      150
## 3 Petal.Length       0         0.00      150
## 4  Petal.Width       0         0.00      150
## 5      Species       0         0.00      150
```

7.7 Comparing two tables at a glance: `compare()`

```
compare(iris, with_na, by = "Species")
```

```
## $dims
## # basetable: 2 x 3
```

```
##   object rows cols
## 1      x  150    5
## 2      y  150    5
##
## $names
## # basetable: 5 x 3
##         column in_x in_y
## 1 Sepal.Length TRUE TRUE
## 2  Sepal.Width TRUE TRUE
## 3 Petal.Length TRUE TRUE
## 4  Petal.Width TRUE TRUE
## 5      Species TRUE TRUE
##
## $types
## # basetable: 5 x 5
##         column class.x typeof.x class.y typeof.y
## 1 Sepal.Length numeric   double numeric   double
## 2  Sepal.Width numeric   double numeric   double
## 3 Petal.Length numeric   double numeric   double
## 4  Petal.Width numeric   double numeric   double
## 5      Species  factor  integer  factor  integer
##
## $missing
## # basetable: 5 x 7
##         column missing.x missing_prop.x complete.x missing.y missing_prop.y complete.y
## 1 Sepal.Length         0              0        150        12            0.08        138
## 2  Sepal.Width         0              0        150         0            0.00        150
## 3 Petal.Length         0              0        150         0            0.00        150
## 4  Petal.Width         0              0        150         0            0.00        150
## 5      Species         0              0        150         0            0.00        150
##
## $key_overlap
## # basetable: 1 x 3
##   x_unique y_unique common
## 1        3        3      3
```

8 Data cleaning and validation

8.1 Assertions that stop on failure: `assert*()`

Each `assert*()` function returns its input invisibly if the check passes, and throws an informative error otherwise, useful as a pipeline guard.

```
mtcars |>
  assertnames(c("mpg", "cyl")) |>
  assertrows(mpg > 0) |>
  assertrange("mpg", lower = 0, upper = 60) |>
  asserttype("cyl", "numeric") |>
  assertcomplete() |>
  headtail(2)
```

```
## # basetable: 4 x 11
##    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## 2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
```

```
## 3 15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
## 4 21.4   4  121 109 4.11 2.780 18.60  1  1    4    2
```

```
tryCatch(assertkey(mtcars, "cyl"), error = function(e) conditionMessage(e))
```

```
## [1] "`by` does not identify unique rows."
```

```
tryCatch(assertunique(mtcars, "cyl"), error = function(e) conditionMessage(e))
```

```
## [1] "Values are not unique."
```

```
tryCatch(assertvalues(mtcars, "am", allowed = c(0, 1)), error = function(e) "passes")
```

`assertcols()`/`assertkey()` are convenience aliases for the lower-level `assert_cols()`/`assert_key()`, which do the same checks and are what you'd call from code that doesn't need the `assert*` naming convention:

```
assert_cols(mtcars, c("mpg", "cyl")) |> invisible()
tryCatch(assert_key(mtcars, "cyl"), error = function(e) conditionMessage(e))
```

```
## [1] "`by` does not identify unique rows."
```

8.2 Finding rows/values that fail a check: `invalidrows()`, `invalidvalues()`, `outofrange()`

These are the non-throwing counterparts of the `assert*()` functions above: instead of stopping, they return the offending rows.

```
invalidrows(mtcars, mpg > 15)
```

```
## # basetable: 6 x 11
##    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1 14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## 2 10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## 3 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
## 4 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## 5 13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
## 6 15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
```

```
outofrange(mtcars, "mpg", lower = 15, upper = 30)
```

```
## # basetable: 9 x 11
##    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## 2 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## 3 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## 4 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## 5 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## 6 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## 7 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## 8 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## 9 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
```

8.3 Missing-data helpers: `missingrows()`, `keepmissing()`, `omitmissing()`, `missingindicator()`

```
d <- data.frame(a = c(1, NA, 3), b = c(NA, NA, 3))
missingrows(d, mode = "any")
```

```
## # basetable: 2 x 2
##    a  b
## 1  1 NA
## 2 NA NA
```

```
keepmissing(d)
```

```
## # basetable: 2 x 2
##    a  b
## 1  1 NA
## 2 NA NA
```

```
omitmissing(d, mode = "any")
```

```
## # basetable: 1 x 2
##   a b
## 1 3 3
```

```
missingindicator(d)
```

```
## # basetable: 3 x 4
##    a  b missing_a missing_b
## 1  1 NA     FALSE      TRUE
## 2 NA NA      TRUE      TRUE
## 3  3  3     FALSE     FALSE
```

8.4 Filling in missing combinations: `completegrid()`, `expandrows()`

`completegrid()` fills in every combination of the named columns that's missing from the data (useful before time-series analysis). `expandrows()` repeats each row a given number of times.

```
sparse <- data.frame(site = c("A", "A", "B"), year = c(2020, 2021, 2020), value = c(1, 2, 3))
completegrid(sparse, cols = c("site", "year"), fill = list(value = 0))
```

```
## # basetable: 4 x 3
##   site year value
## 1    A 2020     1
## 2    B 2020     3
## 3    A 2021     2
## 4    B 2021     0
```

```
expandrows(data.frame(x = c(1, 2)), times = c(2, 1))
```

```
## # basetable: 3 x 1
##   x
## 1 1
## 2 1
## 3 2
```

8.5 Recoding values: `naif()`, `nato()`, `blanktona()`, `natoblank()`, `replacevalues()`, `replacewhere()`, `replacecols()`

```
naif(c(1, 99, 2), 99)
```

```
## [1]  1 NA  2
```

```
nato(c(1, NA, 2), 0)
```

```
## [1] 1 0 2
```

```
blanktona(c("a", "", "b"))
```

```
## [1] "a" NA  "b"
```

```
replacevalues(c("a", "b", "c"), old = c("a", "b"), new = c("A", "B"))
```

```
## [1] "A" "B" "c"
```

```
replacewhere(mtcars, cyl == 4, cols = "mpg", value = NA) |> headtail(2)
```

```
## # basetable: 4 x 11
##   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1  21   6  160 110 3.90 2.620 16.46  0  1    4    4
## 2  21   6  160 110 3.90 2.875 17.02  0  1    4    4
## 3  15   8  301 335 3.54 3.570 14.60  0  1    5    8
## 4  NA   4  121 109 4.11 2.780 18.60  1  1    4    2
```

8.6 Deduplication: `uniquerows()`, `removeduplicates()`, `duplicaterows()`, `duplicatekeys()`, `duplicatenames()`

```
dup_df <- data.frame(id = c(1, 1, 2), v = c("a", "b", "c"))
uniquerows(dup_df, cols = "id")
```

```
## # basetable: 2 x 1
##   id
## 1  1
## 2  2
```

```
removeduplicates(dup_df, by = "id", keep = "first")
```

```
## # basetable: 2 x 2
##   id v
## 1  1 a
## 2  2 c
```

```
duplicaterows(data.frame(x = c(1, 1, 2)))
```

```
## # basetable: 2 x 1
##   x
## 1 1
## 2 1
```

```
duplicatekeys(dup_df, "id")
```

```
## # basetable: 1 x 2
##   id N
## 1  1 2
```

`duplicatekeys()` is a convenience alias for the lower-level `duplicated_keys()`:

```
duplicated_keys(dup_df, "id")
```

```
## # basetable: 1 x 2
##   id N
## 1  1 2
```

8.7 Column-name hygiene: `cleannames()`, `repairnames()`, `renamewith()`, `commonnames()`

```
messy <- data.frame(`First Name` = 1, `2nd_col` = 2, check.names = FALSE)
cleannames(messy)
```

```
## # basetable: 1 x 2
##   first_name 2nd_col
## 1          1       2
```

```
commonnames(mtcars, iris)
```

```
## character(0)
```

`commonnames()` is a convenience alias for the lower-level `common_names()`:

```
common_names(mtcars, iris)
```

```
## character(0)
```

8.8 Filling in missing values within groups: `filldown()`, `fillup()`, `fillboth()`

`filldown()` carries the last non-missing value forward within each group (last observation carried forward); `fillup()` carries the next non-missing value backward; `fillboth()` does both, filling any remaining gaps from either direction.

```
visits <- data.frame(
  id = c(1, 1, 1, 2, 2),
  visit = c(1, 2, 3, 1, 2),
  treatment = c("A", NA, NA, NA, "B")
)
filldown(visits, cols = "treatment", by = "id")
```

```
## # basetable: 5 x 3
##   id visit treatment
## 1  1     1         A
## 2  1     2         A
## 3  1     3         A
## 4  2     1      <NA>
## 5  2     2         B
```

```
fillup(visits, cols = "treatment", by = "id")
```

```
## # basetable: 5 x 3
##   id visit treatment
## 1  1     1         A
## 2  1     2      <NA>
## 3  1     3      <NA>
## 4  2     1         B
## 5  2     2         B
```

```
fillboth(visits, cols = "treatment", by = "id")
```

```
## # basetable: 5 x 3
##   id visit treatment
## 1  1     1         A
## 2  1     2         A
## 3  1     3         A
```

```
## 4  2     1         B
## 5  2     2         B
```

9 Row and column utilities

9.1 Column metadata: colnames(), rownames(), classes(), uniques(), cardinality(), constants(), emptycols()

```
colnames(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
classes(iris)
```

```
## # basetable: 5 x 2
##         column   class
## 1 Sepal.Length numeric
## 2  Sepal.Width numeric
## 3 Petal.Length numeric
## 4  Petal.Width numeric
## 5      Species  factor
```

```
uniques(iris)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##           35           23           43           22            3
```

```
cardinality(iris, cols = "Species")
```

```
## Species
##       3
```

```
constants(data.frame(a = 1, b = 1:2))
```

```
## [1] "a"
```

```
emptycols(data.frame(a = c(NA, NA), b = 1:2))
```

```
## [1] "a"
```

9.2 Finding blank rows: emptyrows()

```
emptyrows(data.frame(a = c(NA, "x", ""), b = c(NA, "y", NA)))
```

```
##      a    b
## 1 <NA> <NA>
## 3      <NA>
```

9.3 Reordering columns: move(), firstcols(), lastcols()

```
move(mtcars, "hp", before = "mpg") |> colnames()
```

```
##  [1] "hp"   "mpg"  "cyl"  "disp" "drat" "wt"   "qsec" "vs"   "am"   "gear" "carb"
```

```
firstcols(mtcars, "wt") |> colnames()
```

```
##  [1] "wt"   "mpg"  "cyl"  "disp" "hp"   "drat" "qsec" "vs"   "am"   "gear" "carb"
```

```
lastcols(mtcars, "mpg") |> colnames()
```

```
##  [1] "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear" "carb" "mpg"
```

9.4 Row subsets: `firstrows()`, `lastrows()`, `samplerows()`, `samplefrac()`, `reverse()`

```
firstrows(mtcars, 3)
```

```
## # basetable: 3 x 11
##    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## 2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## 3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
lastrows(mtcars, 2)
```

```
## # basetable: 2 x 11
##    mpg cyl disp  hp drat   wt qsec vs am gear carb
## 1 15.0   8  301 335 3.54 3.57 14.6  0  1    5    8
## 2 21.4   4  121 109 4.11 2.78 18.6  1  1    4    2
```

```
set.seed(1)
samplerows(mtcars, 2)
```

```
## # basetable: 2 x 11
##    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1 19.2   8  400 175 3.08 3.845 17.05  0  0    3    2
## 2 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
```

`samplerows()` and `samplefrac()` take `by` to sample within each group (`n` is capped at the group size); sampled rows keep their original order.

```
set.seed(1)
samplerows(mtcars, 2, by = "cyl")
```

```
## # basetable: 6 x 11
##    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## 2 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## 3 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## 4 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## 5 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## 6 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
```

For an arbitrary set of row positions, index the table directly, the same way you would any base R data frame:

```
firstrows(mtcars, c(1, 3, 5))
```

```
## # basetable: 1 x 11
##   mpg cyl disp  hp drat   wt  qsec vs am gear carb
## 1  21   6  160 110  3.9 2.62 16.46  0  1    4    4
```

9.5 Sorting rows: `orderrows()`

```
orderrows(mtcars, by = "mpg", decreasing = TRUE) |> headtail(2)
```

```
## # basetable: 4 x 11
##    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## 2 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## 3 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## 4 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
```

9.6 Row position helpers: `rownumber()`

```
rownumber(c("a", "b", "c"))
```

```
## [1] 1 2 3
```

9.7 Row-wise reductions across columns: `rowmin()`, `rowmax()`, `rowany()`, `rowall()`, `rowcount()`, `rowfirst()`, `rowlast()`, `rowapply()`

```
d <- data.frame(x = c(1, NA, 3), y = c(4, 5, NA), z = c(TRUE, FALSE, TRUE))
rowmin(d, cols = c("x", "y"), na.rm = TRUE)
```

```
## [1] 1 5 3
```

```
rowmax(d, cols = c("x", "y"), na.rm = TRUE)
```

```
## [1] 4 5 3
```

```
rowfirst(d, cols = c("x", "y"), na.rm = TRUE)
```

```
## [1] 1 5 3
```

```
rowlast(d, cols = c("x", "y"), na.rm = TRUE)
```

```
## [1] 4 5 3
```

```
rowapply(d, cols = c("x", "y"), fun = function(row) sum(row, na.rm = TRUE))
```

```
## [1] 5 5 3
```

9.8 Applying a function to selected columns: `applycols()`, `convertcols()`

```
applycols(mtcars, cols = "mpg", fun = round) |> headtail(2)
```

```
## # basetable: 4 x 11
##   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1  21   6  160 110 3.90 2.620 16.46  0  1    4    4
## 2  21   6  160 110 3.90 2.875 17.02  0  1    4    4
## 3  15   8  301 335 3.54 3.570 14.60  0  1    5    8
## 4  21   4  121 109 4.11 2.780 18.60  1  1    4    2
```

```
convertcols(mtcars, cols = "cyl", fun = as.character) |> types()
```

```
## # basetable: 11 x 3
##    column     class    typeof
## 1     mpg   numeric    double
## 2     cyl character character
## 3    disp   numeric    double
## 4      hp   numeric    double
## 5    drat   numeric    double
```

```
## 6      wt   numeric    double
## 7    qsec   numeric    double
## 8      vs   numeric    double
## 9      am   numeric    double
## 10   gear   numeric    double
## # 1 more rows
```

10 String manipulation

10.1 Whitespace and case: `trim()`, `squish()`, `lower()`, `upper()`, `titlecase()`, `sentencecase()`, `textlen()`

```
trim("  hello  ")
```

```
## [1] "hello"
```

```
squish("  too   many   spaces ")
```

```
## [1] "too many spaces"
```

```
lower("HELLO"); upper("hello")
```

```
## [1] "hello"
```

```
## [1] "HELLO"
```

```
titlecase("hello world"); sentencecase("HELLO WORLD")
```

```
## [1] "Hello World"
```

```
## [1] "Hello world"
```

```
textlen(c("abc", NA))
```

```
## [1]  3 NA
```

10.2 Substrings and padding: `left()`, `right()`, `middle()`, `truncate()`, `padleft()`, `padright()`, `padcenter()`

```
left("hello", 3); right("hello", 3); middle("hello", 2, 4)
```

```
## [1] "hel"
```

```
## [1] "llo"
```

```
## [1] "ell"
```

```
truncate("a long sentence", 10)
```

```
## [1] "a long ..."
```

```
padleft("7", 3, pad = "0"); padright("7", 3, pad = "0"); padcenter("hi", 6, pad = "*")
```

```
## [1] "007"
```

```
## [1] "700"
```

```
## [1] "**hi**"
```

10.3 Pattern matching: containstext(), matchestext(), startswith(), endswith(), countmatch(), locate(), locateall()

```
containstext(c("apple", "banana"), "an")
```

```
## [1] FALSE  TRUE
```

```
matchestext(c("cat", "cats"), "cat")
```

```
## [1]  TRUE FALSE
```

```
startswith(c("apple", "banana"), "a")
```

```
## [1]  TRUE FALSE
```

```
endswith(c("apple", "banana"), "a")
```

```
## [1] FALSE  TRUE
```

```
countmatch("banana", "a")
```

```
## [1] 3
```

```
locate("banana", "an")
```

```
## [1] 2
## attr(,"match.length")
## [1] 2
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

10.4 Extraction: extract(), extractall(), extractnum(), extractint(), extractbetween()

```
extract("order #4231", "[0-9]+")
```

```
## [1] "4231"
```

```
extractall("a1b2c3", "[0-9]")
```

```
## [[1]]
## [1] "1" "2" "3"
```

```
extractnum("total: -12.5 kg")
```

```
## [1] -12.5
```

```
extractbetween("[important]", "\\[", "\\]")
```

```
## [1] "important"
```

10.5 Replace and remove: replacetext(), removetext(), removeall(), replaceall()

```
replacetext("a-b-c", "-", "_")
```

```
## [1] "a_b_c"
```

```r
removetext("a-b-c", "-")
```

```
## [1] "abc"
```

```r
replaceall(c("a", "b"), old = "a", new = "A")
```

```
## [1] "A" "b"
```

10.6 Split and join: splittext(), splitfirst(), splitlast(), jointext(), collapsetext()

```r
splittext("a-b-c", "-")
```

```
## [[1]]
## [1] "a" "b" "c"
```

```r
splitfirst("a-b-c", "-"); splitlast("a-b-c", "-")
```

```
## [1] "a"
```

```
## [1] "c"
```

```r
jointext("a", "b", "c")
```

```
## [1] "abc"
```

```r
collapsetext(c("a", "b", "c"), sep = ", ")
```

```
## [1] "a, b, c"
```

10.7 Blank and encoding helpers: isblank(), removeaccents(), normalizeunicode(), normalizeencoding(), transliterate()

```r
isblank(c("", "  ", NA, "x"))
```

```
## [1]  TRUE  TRUE  TRUE FALSE
```

```r
removeaccents("café")
```

```
## [1] "cafe"
```

10.8 String distance and similarity: textdist(), nearesttext(), similartext()

```r
textdist("cat", c("cat", "bat", "dog"))
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    3
```

```r
nearesttext("kat", c("cat", "dog", "bird"))
```

```
## [1] "cat"
```

10.9 Classification predicates: isalpha(), isalphanumeric(), isnumerictext(), isintegertext(), isemail(), isurl()

```r
isalpha(c("abc", "a1c"))
```

```
## [1]  TRUE FALSE
```

```r
isnumerictext(c("12.5", "abc"))
```

```
## [1]  TRUE FALSE
```

```r
isemail(c("a@b.com", "not-an-email"))
```

```
## [1]  TRUE FALSE
```

```r
isurl(c("https://example.com", "example.com"))
```

```
## [1]  TRUE FALSE
```

10.10 Recoding and grouping values: `recode()`, `collapsevalues()`, `casewhen()`, `collapselevels()`, `lump()`, `reorderlevels()`, `expandlevels()`

```r
recode(c("a", "b", "c"), old = "a", new = "A")
```

```
## [1] "A" "b" "c"
```

```r
collapsevalues(c("cat", "dog", "bird"), groups = list(pet = c("cat", "dog")))
```

```
## [1] "pet"  "pet"  "bird"
```

```r
x <- rep(c("a", "b", "c", "d"), c(10, 5, 3, 1))
lump(x, n = 2, other = "Other") |> table()
```

```
##
## Other     a     b
##     4    10     5
```

```r
f <- factor(c("low", "high", "mid"), levels = c("low", "mid", "high"))
reorderlevels(f, by = c("high", "mid", "low"))
```

```
## [1] low  high mid
## Levels: high mid low
```

```r
expandlevels(f, "extra")
```

```
## [1] low  high mid
## Levels: low mid high extra
```

`casewhen()` maps conditions to a label, taking the same named-list shape as `collapsevalues()` but with logical vectors: the first element that is `TRUE` at a position gives its value there, and `default` covers the rest. It composes inside `transform()`.

```r
transform(mtcars, size = casewhen(
  list(light = wt < 2.5, mid = wt < 3.5),
  default = "heavy"
)) |> pick(c("wt", "size")) |> headtail(2)
```

```
## # basetable: 4 x 2
##      wt  size
## 1 2.620   mid
## 2 2.875   mid
## 3 3.570 heavy
## 4 2.780   mid
```

11 Parsing text into typed values

`parse*()` functions convert messy text into typed vectors, with an `na=` for sentinel missing-value strings and a `strict=` to error instead of silently returning `NA` on failure.

```
parseint(c("1", "2", "NA"))
```

```
## [1]  1  2 NA
```

```
parsenum("1.234,56", decimal = ",", grouping = ".")
```

```
## [1] 1235
```

```
parselogical(c("TRUE", "false", "T", "0"))
```

```
## [1]  TRUE FALSE  TRUE FALSE
```

```
parsedate(c("2024-01-15", "15/01/2024"), formats = c("%Y-%m-%d", "%d/%m/%Y"))
```

```
## [1] "2024-01-15" "2024-01-15"
```

```
parsedatetime("2024-01-15 08:30:00", formats = "%Y-%m-%d %H:%M:%S")
```

```
## [1] "2024-01-15 08:30:00 UTC"
```

```
parsepercent("42%")
```

```
## [1] 0.42
```

```
parsecurrency("$1,234.50")
```

```
## [1] 1234
```

```
parsefailures(c("1", "x", "3"), parseint)
```

```
## # basetable: 1 x 2
##   index value
## 1     2     x
```

12 Dates and times

12.1 Extracting components

```
d <- as.Date("2024-03-15")
year(d); month(d); day(d); weekday(d); yearday(d); week(d); quarter(d)
```

```
## [1] 2024
```

```
## [1] 3
```

```
## [1] 15
```

```
## [1] "Friday"
```

```
## [1] 75
```

```
## [1] 10
```

```
## [1] 1
```

```
t <- as.POSIXct("2024-03-15 08:30:45", tz = "UTC")
hour(t); minute(t); second(t)
```

```
## [1] 8
```

```
## [1] 30
```

```
## [1] 45
```

12.2 Arithmetic and sequences: `adddays()`, `addweeks()`, `addmonths()`, `addyears()`, `datediff()`, `dateseq()`, `betweendates()`

`addmonths()`/`addyears()` handle end-of-month overflow explicitly via `invalid = c("previous", "next", "missing", "error")`: e.g. what should January 31st plus one month become, given February doesn't have 31 days?

```
adddays(d, 10); addweeks(d, 2)
```

```
## [1] "2024-03-25"
```

```
## [1] "2024-03-29"
```

```
addmonths(as.Date("2024-01-31"), 1, invalid = "previous")
```

```
## [1] "2024-02-29"
```

```
addmonths(as.Date("2024-01-31"), 1, invalid = "next")
```

```
## [1] "2024-03-02"
```

```
datediff(as.Date("2024-01-10"), as.Date("2024-01-01"), units = "days")
```

```
## [1] 9
```

```
dateseq(as.Date("2024-01-01"), as.Date("2024-01-05"))
```

```
## [1] "2024-01-01" "2024-01-02" "2024-01-03" "2024-01-04" "2024-01-05"
```

```
betweendates(d, "2024-01-01", "2024-12-31")
```

```
## [1] TRUE
```

12.3 Rounding dates: `floordate()`, `ceilingdate()`, `rounddate()`

```
floordate(d, "month"); ceilingdate(d, "month"); rounddate(d, "month")
```

```
## [1] "2024-03-01"
```

```
## [1] "2024-04-01"
```

```
## [1] "2024-03-01"
```

```
floordate(d, "week")
```

```
## [1] "2024-03-11"
```

13 Numeric transforms and rolling windows

13.1 Rescaling and standardizing: `rescale()`, `standardize()`, `center()`, `winsorize()`

```
x <- c(1, 2, 3, 4, 100)
rescale(x); standardize(x); center(x)
```

```
## [1] 0.0000 0.0101 0.0202 0.0303 1.0000
```

```
## [1] -0.4815 -0.4585 -0.4356 -0.4127  1.7883
```

```
## [1] -21 -20 -19 -18  78
```

```
winsorize(x, probs = c(0.1, 0.9))
```

```
## [1]  1.4  2.0  3.0  4.0 61.6
```

13.2 Binning: `quantilegroup()`

```
quantilegroup(1:100, n = 4) |> table()
```

```
##
##    [1,25.8] (25.8,50.5] (50.5,75.2]  (75.2,100]
##          25          25          25          25
```

13.3 Period-over-period change and rank: `percentchange()`, `percentrank()`, `denserank()`

`denserank()` ranks values by sorted order with no gaps between ranks, tied values sharing a rank (the same semantics as SQL's `DENSE_RANK()`).

```
percentchange(c(100, 110, 99))
```

```
## [1]  NA  10 -10
```

```
percentrank(c(10, 20, 20, 30))
```

```
## [1] 0.250 0.625 0.625 1.000
```

```
denserank(c(30, 10, 20, 10, 30))
```

```
## [1] 3 1 2 1 3
```

13.4 Cumulative helpers: `cumcount()`, `cumedist()`, `cumavg()`

```
cumcount(c("a", "b", "c"))
```

```
## [1] 1 2 3
```

```
cumedist(c("a", "a", "b"))
```

```
## [1] 1.0000 0.5000 0.6667
```

```
cumavg(c(1, 2, 3))
```

```
## [1] 1.0 1.5 2.0
```

13.5 Lag/lead and differencing: `difference()`, `lagvalue()`, `leadvalue()`

```
difference(c(1, 3, 6, 10))
```

```
## [1] NA  2  3  4
```

```
lagvalue(1:5, n = 1)
```

```
## [1] NA  1  2  3  4
```

```
leadvalue(1:5, n = 1)
```

```
## [1]  2  3  4  5 NA
```

13.6 Rolling window functions: `rollmean()`, `rollsum()`, `rollmin()`, `rollmax()`, `rollmedian()`, `rollsd()`, `rollvar()`, `rollprod()`, `rollapply()`

All roll functions share the same interface: a `width`, an `align` ("right"/"left"/"center"), a `fill` for incomplete windows, `partial` to allow incomplete boundary windows instead of `fill`, and (where numerically meaningful) `na.rm`.

```
v <- c(1, 2, NA, 4, 5)
rollmean(v, width = 3, na.rm = TRUE)
```

```
## [1]  NA  NA 1.5 3.0 4.5
```

```
rollsum(v, width = 3, na.rm = TRUE, partial = TRUE)
```

```
## [1] 1 3 3 6 9
```

```
rollmax(v, width = 2, na.rm = TRUE)
```

```
## [1] NA  2  2  4  5
```

```
rollapply(1:5, width = 3, FUN = sum, partial = TRUE)
```

```
## [1]  1  3  6  9 12
```

14 Functional helpers

`map()`/`traverse()`/`foldr()` are lightweight base-R helpers for working with plain vectors and lists (not tables), useful for the occasional loop-replacement inside a larger `basetable` pipeline.

```
map(1:3, function(x) x + 0.5)
```

```
## [[1]]
## [1] 1.5
##
## [[2]]
## [1] 2.5
##
## [[3]]
## [1] 3.5
```

```
traverse(list(a = 1:2, b = 10:11), function(a, b) a + b)
```

```
## [[1]]
## [1] 11
##
## [[2]]
## [1] 13
```

```
foldr(1:4, `+`)
```

```
## [1] 10
```

15 Closing notes

This document covers all of `basetable`'s exported functions. For the performance story (how basetable compares to base R and dplyr, and where its overhead was tracked down and fixed), see the companion `benchmarking` vignette.