Package 'mixtime'

October 13, 2026

Title Mixed Temporal Vectors and Operations

Version 0.4.0

Description Flexible time classes for time series analysis and forecasting with mixed temporal granularities. Supports linear and cyclical time representations in discrete and continuous forms, with timezone support, across multiple calendar systems including Gregorian and ISO week date calendars. Time points are stored numerically relative to a chronon; an atomic time granule defined by time units of a calendar. Calendrical arithmetic enables conversion between time granules (e.g. days to months) and calendar systems. Multi-unit arithmetic allows for temporal analysis with other granules of common calendars (e.g. fortnights are 2-week units). Time vectors of different granularities (e.g. monthly and quarterly) can be combined in a single vector, making 'mixtime' ideal for data that changes observation frequency over time or requires temporal reconciliation across scales. The package is extensible, allowing users to define custom calendars that build upon civil and astronomical time systems.

License MIT + file LICENSE

URL https://pkg.mitchelloharawild.com/mixtime/, https://github.com/mitchelloharawild/mixtime

BugReports https://github.com/mitchelloharawild/mixtime/issues

Encoding UTF-8

Language en-GB

Depends R (>= 3.0.2)

Imports lifecycle, vctrs, rlang, cli, S7, vecvec (>= 1.2.0), tzdb, methods

Suggests stats, tsibble, testthat, pillar, knitr, rmarkdown, ggplot2, ggtime

LinkingTo cpp11 (>= 0.5.2), tzdb (>= 0.5.0)

RdMacros lifecycle

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Mitchell O'Hara-Wild [aut, cre] (ORCID:
<https://orcid.org/0000-0001-6729-7695>),
Cynthia Huang [ctb] (ORCID: <https://orcid.org/0000-0002-9218-987X>),
Rob Hyndman [ctb] (ORCID: <https://orcid.org/0000-0002-2140-5352>)

Maintainer Mitchell O'Hara-Wild <mail@mitchelloharawild.com>

Repository CRAN

Date/Publication 2026-10-13 03:40:02 UTC

Contents

allen-interval-algebra . . . 3
as_mixtime . . . 6
calendar_gregorian . . . 6
calendar_isoweek . . . 8
calendar_sym454 . . . 9
calendar_time_civil . . . 10
calendar_time_lunar . . . 11
calendar_time_solar . . . 12
chronon_cardinality . . . 14
chronon_divmod . . . 15
chronon_epoch . . . 16
chronon_format_attr . . . 16
chronon_format_linear . . . 17
chronon_glb . . . 18
chronon_parse_linear . . . 19
circsum . . . 20
class_mixtime . . . 21
component_helpers . . . 22
cyclical_time . . . 23
cyclical_time_helpers . . . 25
duration . . . 26
duration_helpers . . . 27
format.mt_time . . . 29
is_mixtime . . . 29
is_time . . . 30
label_scheme . . . 31
linear_labels_format . . . 35
linear_time . . . 37
linear_time_helpers . . . 38
mixtime . . . 40
mixtime_location . . . 42
mt_cyclical-compare . . . 42
mt_duration-compare . . . 44
mt_linear-compare . . . 45
mt_time-class . . . 46

mt_unit . . . 47
new_calendar . . . 48
new_cyclical_time_fn . . . 50
new_duration_fn . . . 51
new_linear_time_fn . . . 52
new_mixtime . . . 53
new_time . . . 53
new_time_unit . . . 54
parse_format . . . 55
seq.mixtime::mixtime . . . 56
time_calendar . . . 59
time_chronon . . . 60
time_components . . . 61
time_compose . . . 62
time_cycle . . . 63
time_is_complete_at . . . 64
time_is_determinate_at . . . 65
time_parse . . . 66
time_round . . . 68
time_shift . . . 69
time_unit_full . . . 71
tz_abbreviation . . . 72
tz_name . . . 72
tz_offset . . . 73
tz_transitions . . . 73
vocab_table . . . 74

Index **76**

`allen-interval-algebra`

Allen's interval algebra for time vectors

Description

Allen's interval algebra has thirteen base relations for time intervals. `==` is defined in mt_linear-compare/mt_cyclical-compare; the twelve operators below define the rest, computed from the same interval endpoints (`start`/`end`) that comparison derives. Together, they cover all thirteen relations:

Operator	Relation	`e1 %op% e2` holds iff
`%p%`	precedes	`e1` ends before `e2` begins (a gap)
`%m%`	meets	`e1` ends exactly where `e2` begins (no gap, no overlap)
`%o%`	overlaps	`e1` begins before `e2`, `e1` ends between the start and end of `e2`
`%s%`	starts	`e1` and `e2` begin together, and `e1` ends first
`%d%`	during	`e1` is strictly within `e2`'s span
`%f%`	finishes	`e1` and `e2` end together, and `e1` begins later
`==`	equals	`e1` and `e2` share both endpoints

`%pi%`	preceded by	`e2 %p% e1`
`%mi%`	met by	`e2 %m% e1`
`%oi%`	overlapped by	`e2 %o% e1`
`%si%`	started by	`e2 %s% e1`
`%di%`	contains	`e2 %d% e1`
`%fi%`	finished by	`e2 %f% e1`

mt_linear-compare/mt_cyclical-compare's </> are a deliberate deviations from their use as 'precedes' in Allen's interval algebra (`%p%`/`%pi%` operators). The important difference is that </> include 'meets' relations, while `%p%`/`%pi%` requires gaps, so in the examples below `jan < feb` is TRUE in mixtime but the equivalent symbol in Allen's interval algebra (`jan %p% feb`) is FALSE because there is no gap.

The implementation here also extends to continuous time instants, which are treated as degenerate zero-width intervals. This allows testing how a specific time instant relates to time spans, for example does January 2020 contain 30% through the day 2020-01-24 is `yearmonth("2020-01") %di% date(datetime("2020-01-24 07:12:00"), discrete = FALSE)`. This is a technical deviation from Allen's interval algebra, which assumes that intervals are non-degenerate (i.e. end > start, not end >= start).

Usage

```
e1 %p% e2

e1 %pi% e2

e1 %m% e2

e1 %mi% e2

e1 %o% e2

e1 %oi% e2

e1 %s% e2

e1 %si% e2

e1 %d% e2

e1 %di% e2

e1 %f% e2

e1 %fi% e2
```

Arguments

`e1, e2`	`mt_linear` or `mt_cyclical` vectors sharing a mode of time (and, for `mt_cyclical`,

a cycle) - or values castable to one, such as plain numeric vectors in the other operand's chronon, read from its epoch as `mixtime()` reads them. `mt_duration` has no interval to compare and so is not supported.

Value

A logical vector.

References

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11), 832-843.

See Also

mt_linear-compare and mt_cyclical-compare for ==, <, >, <=, >= - the three of Allen's relations that also form the package's ordering (</> in a looser, adjacency-inclusive form than `%p%`/`%pi%`).

Examples

```
jan <- yearmonth("2020 Jan")
feb <- yearmonth("2020 Feb")
mar <- yearmonth("2020 Mar")
q1 <- yearquarter("2020 Q1")

jan == jan    # January equals itself

jan %p% mar   # January precedes March: February leaves a genuine gap
mar %pi% jan  # ... equivalently, March is preceded by January

jan %m% feb   # January meets February: adjacent, no gap
feb %mi% jan  # ... equivalently, February is met by January
jan < feb     # `<` also holds for adjacent pairs, unlike `%p%`
jan %p% feb   # FALSE: no gap between them, so they don't (strictly) precede

jan %s% q1    # January and Q1 start together, but January finishes first
q1 %si% jan   # ... equivalently, Q1 is started by January

feb %d% q1    # February is strictly within Q1
q1 %di% feb   # ... equivalently, Q1 contains February

mar %f% q1    # March and Q1 finish together, but March starts later
q1 %fi% mar   # ... equivalently, Q1 is finished by March
```

as_mixtime *Convert a time class into a mixtime*

Description

Coerces a time object (e.g. `Date`, `POSIXct`, `yearmonth`) to a `mixtime` vector using `vctrs::vec_cast()`. The chronon and cycle are inferred from `x` via `time_chronon()` and `time_cycle()`.

Usage

```
as_mixtime(x, ...)
```

Arguments

`x`	A time value to convert to a `mixtime`. Any time class with a defined `time_chronon()` method can be converted (e.g. `Date`, `POSIXct`, `yearmonth`, etc.).
`...`	Additional arguments passed to the underlying `vec_cast()` method.

Value

A `mixtime` object corresponding to `x`.

See Also

`mixtime()` for constructing a `mixtime` directly from data, `is_mixtime()` for testing if an object is a `mixtime`.

Examples

```
as_mixtime(Sys.Date())
as_mixtime(Sys.time())
```

calendar_gregorian *Gregorian time unit classes*

Description

Time unit constructors for the Gregorian calendar system. These units can be used with `linear_time()` to create custom time representations.

Usage

```
cal_gregorian
```

Format

A civil-based calendar containing Gregorian time units.

Details

The following time units are available in the Gregorian calendar (`cal_gregorian$`).

- `year()`: Year unit
- `quarter()`: Quarter (3-month period) unit
- `month()`: Month unit
- `day()`: Day unit
- `hour()`: Hour unit
- `minute()`: Minute unit
- `second()`: Second unit
- `millisecond()`: Millisecond unit

These units form a hierarchy where conversions between adjacent units follow the Gregorian calendar rules. For units that don't have a fixed relationship (e.g., months to days), the conversion requires a time context.

Value

An S3 list of class `c("cal_gregorian", "mt_calendar")` containing the named time unit classes of the Gregorian calendar. Each unit is accessible via `$` notation and calling it with a step size produces a time granule (e.g., 1 month granule as `cal_gregorian$month(1L)`).

See Also

`linear_time()` for creating linear time points.

Examples

```
# Create a custom time representation using Gregorian units
linear_time(
  Sys.time(),
  chronon = hour(1L)
)
```

`calendar_isoweek` *ISO 8601 time unit classes*

Description

Time unit constructors for the ISO 8601 calendar system. These units can be used with `linear_time()` to create custom time representations.

Usage

```
cal_isoweek
```

Format

A civil-based calendar containing ISO 8601 time units.

Details

The following time units are available in the ISO week date calendar:

- `year()`: ISO year unit (years start on the week containing the first Thursday)
- `week()`: Week unit (7-day periods)
- `day()`: Day unit
- `hour()`: Hour unit
- `minute()`: Minute unit
- `second()`: Second unit
- `millisecond()`: Millisecond unit

ISO 8601 weeks always start on Monday and the first week of a year is the week containing the first Thursday of that year. This means that some days in early January may belong to the last week of the previous ISO year, and some days in late December may belong to the first week of the next ISO year.

Value

An S3 list of class `c("cal_isoweek", "mt_calendar")` containing the named time unit classes of the ISO 8601 week calendar. Each unit is accessible via `$` notation and calling it with a step size produces a time granule (e.g., 1 week granule as `cal_isoweek$week(1L)`).

See Also

`linear_time()` for creating custom time representations, `yearweek()` for a pre-defined ISO 8601 year-week representation

`calendar_sym454` *Symmetry454 time unit classes*

Description

Time unit constructors for the Symmetry454 calendar system. These units can be used with `linear_time()` to create custom time representations.

Usage

```
cal_sym454
```

Format

A civil-based calendar containing Symmetry454 time units.

Details

The Symmetry454 calendar (Sym454) is a perennial solar calendar proposed by Dr. Irv Bromberg. It preserves the traditional 12-month structure and 7-day week, with months arranged in a symmetrical 4-5-4 week pattern per quarter. Every month starts on Monday and has a whole number of weeks, meaning no month ever contains a partial week.

The following time units are available in the Symmetry454 calendar (`cal_sym454$`).

- `year()`: Year unit
- `month()`: Month unit
- `week()`: Week unit
- `day()`: Day unit
- `hour()`: Hour unit
- `minute()`: Minute unit
- `second()`: Second unit
- `millisecond()`: Millisecond unit

Leap years:

Rather than intercalary days, Symmetry454 uses a **leap week** appended to December once every 5 or 6 years, making December a 5-week month in leap years. A year is a leap year if `(52 * year + 146) %% 293 < 52`.

This yields a mean year of 365 + 71/293 days (approx 365 days, 5 hours, 48 minutes, 56.5 seconds), intentionally slightly shorter than the mean northward equinoctial year.

Value

An S3 list of class `c("cal_sym454", "mt_calendar")` containing the named time unit classes of the Symmetry454 calendar. Each unit is accessible via `$` notation and calling it with a step size produces a time granule (e.g., 1 week granule as `cal_sym454$week(1L)`).

See Also

linear_time() for creating linear time points, and https://en.wikipedia.org/wiki/Symmetry454 for more calendar details.

Examples

```
# Create a custom time representation using Symmetry454 units
linear_time(
  Sys.time(),
  chronon = cal_sym454$week(1L)
)
```

---

calendar_time_civil *Civil time unit classes*

Description

Time unit constructors for the civil time system where the boundary of each day is at midnight on the 24 hour clock. This calendar is intended to be built on by other calendars (e.g. [cal_time_civil] and [cal_isoweek]) to add common time components. These units can be used with linear_time() to create custom time representations.

Usage

```
cal_time_civil
```

Details

The following time units are available (cal_time_civil$).

- day(): Day unit
- hour(): Hour unit
- minute(): Minute unit
- second(): Second unit
- millisecond(): Millisecond unit
- microsecond(): Microsecond unit
- nanosecond(): Nanosecond unit

Value

A time granule object for the civil time system.

See Also

cal_time_civil, cal_isoweek

Examples

```
# Create a custom time representation using civil time granules
hms <- new_cyclical_time_fn(
  chronon = second(1L),
  cycle = hour(1L)
)
```

`calendar_time_lunar` *Lunar time unit classes*

Description

Time unit constructors for the synodic lunar time system, where the boundary of each month is at the new moon (lunar conjunction) and the boundary of each phase is at a lunar octant (each eighth of the synodic cycle). Both boundaries are location-independent, as the new moon is a geocentric astronomical event. `cal_time_lunar` is an alias for `cal_time_lunar_synodic`.

Usage

```
cal_time_lunar_synodic

cal_time_lunar
```

Format

A calendar containing synodic lunar time units.

Details

The following time units are available in the lunar calendar systems.

- `month()`: Synodic month unit
- `phase()`: Synodic phase unit

Value

An S3 list of class `c("cal_time_lunar", "mt_calendar")` containing the named time unit classes of the synodic lunar calendar. Each unit is accessible via `$` notation and calling it with a step size produces a time granule (e.g., 1 synodic month granule as `cal_time_lunar$month(1L)`). Lunar month and phase boundaries are location-independent.

See Also

`cal_time_civil`, `cal_time_solar`

Examples

```
# Find the time of the most recent new moon in the Gregorian calendar
time_floor(datetime(Sys.time(), tz = "Australia/Melbourne"), cal_time_lunar$month(1L))
```

`calendar_time_solar` *Solar time unit classes*

Description

Time unit constructors for the transit-based solar time system, where the boundary of each day is at apparent solar midnight. Solar events define the `ampm` (midnight and noon) and `illumination` (dawn, sunrise, sunset, dusk) units. `cal_time_solar` is an alias for `cal_time_solar_transit`.

Usage

```
cal_time_solar_transit

cal_time_solar
```

Format

A location-based calendar containing transit-based solar time units.

Details

The following time units are available in the solar calendar systems.

- `day()`: Day unit
- `ampm()`: Half-day units (AM = before solar noon, PM = after solar noon)
- `hour()`: Hour units within the solar day
- `minute()`: Minute units within the solar hour
- `second()`: Second units within the solar minute
- `degree()`: Solar angle units within the day
- `arcminute()`: Arcminute units within the solar degree
- `arcsecond()`: Arcsecond units within the solar arcminute
- `illumination()`: Illumination phases (night, astronomical/nautical/civil dawn, day, civil/nautical/astronomical dusk)

AM/PM half-days:

The `ampm` unit divides each solar day into two halves between solar noon and solar midnight:

Half	Period	Description
AM	Solar midnight to noon	Morning half; before solar transit

PM Solar noon to midnight Afternoon half; after solar transit

Solar illumination phases:

Phases describe the illumination state of the sky and correspond to standard twilight definitions used in astronomy and navigation. Each phase is bounded by a pair of solar altitude thresholds:

Phase	Solar altitude range	Description
Night	< -18°	Sky fully dark; from last dusk to first dawn (spans noon)
Astronomical dawn	-18° to -12°	Astronomical twilight before sunrise; faint objects obscured
Nautical dawn	-12° to -6°	Nautical twilight before sunrise; horizon visible at sea
Civil dawn	-6° to -0.833°	Civil twilight before sunrise; sky brightening in the east
Day	> -0.833°	Sun above the horizon; spans solar noon
Civil dusk	-0.833° to -6°	Civil twilight after sunset; sky fading in the west
Nautical dusk	-6° to -12°	Nautical twilight after sunset; horizon visible at sea
Astronomical dusk	-12° to -18°	Astronomical twilight after sunset; faint objects obscured

The -0.833° threshold for sunrise and sunset accounts for the mean angular radius of the solar disc (0.267°) plus the standard atmospheric refraction at the horizon (0.566°). Noon and midnight are derived from the equation of time rather than a fixed altitude. Locations that experience polar day or polar night (civil days where sunrise does not occur) are not currently supported, it is recommended to use an alternative reference location.

Value

An S3 list of class `c("cal_time_solar", "mt_calendar")` containing the named time unit classes of the solar transit calendar. Each unit is accessible via `$` notation and calling it with a step size and location produces a time granule (e.g., 1 solar day granule as `cal_time_solar$day(1L, lat = 0, lon = 0)`). Because solar day boundaries depend on the observer's position, each unit constructor requires `lat` and `lon` arguments.

See Also

`cal_time_civil`, `cal_time_lunar`

Examples

```
# Find the current solar time in Melbourne
datetime(Sys.time(), calendar = cal_time_solar, lat = -37.8136, lon = 144.9631)
```

`chronon_cardinality` *Cardinality between time granules*

Description

This S7 generic function defines the calendrical relationships between two chronons, and is one of the building block for defining calendars in mixtime. It calculates how many `x` chronons fit into the `y` chronon. Some chronon sizes are context-dependent (such as the number of days in a month), and so an optional time point defined in terms of `y` chronons can be provided with `at`.

Usage

```
chronon_cardinality(x, y, ...)
```

Arguments

`x`	The finer time granule (e.g. `cal_gregorian$month(1L)`)
`y`	The coarser time granule (e.g. `cal_gregorian$year(1L)`)
`...`	Additional arguments for methods.

Details

The methods are dispatched based on the shortest path along defined methods. This allows for defining only the direct relationships between adjacent time units, and relying on graph traversal to find how to convert between more distant units. For example the number of seconds in an hour can be calculated from the number of seconds in a minute and then number of minutes in an hour.

If a method is defined for converting between time units of different calendar systems (e.g., Gregorian calendar days to Chinese calendar days), then that method can be used to convert times at any granularity between the two systems.

Value

Numeric describing how many `x` time granules fit into `y` at time `at`.

Examples

```
# There are 12 months in a year
with(cal_gregorian, chronon_cardinality(month(1L), year(1L)))

# There are 7 days in a week
with(cal_isoweek, chronon_cardinality(day(1L), week(1L)))

# There are 3600 seconds in an hour
with(cal_gregorian, chronon_cardinality(second(1L), hour(1L)))

# There are 18 "2 months" in 3 years
with(cal_gregorian, chronon_cardinality(month(2L), year(3L)))
```

```
# There are 365 days in 2025 (a common year)
chronon_cardinality(
  cal_gregorian$day(1L), cal_gregorian$year(1L),
  at = year(as.Date("2025-01-01"))
)

# There are 366 days in 2024 (a leap year)
chronon_cardinality(
  cal_gregorian$day(1L), cal_gregorian$year(1L),
  at = mixtime::year(as.Date("2024-01-01"))
)

# There are 29 days in February 2024 (a leap year)
chronon_cardinality(
  cal_gregorian$day(1L), cal_gregorian$month(1L),
  at = yearmonth(as.Date("2024-02-01"))
)
```

`chronon_divmod` *Convert between chronons of different time granules*

Description

This function converts between chronons measured in different time granules. It is used internally for converting between different continuous time types, and is particularly useful for efficiently converting between irregular time granules. The default method uses `chronon_cardinality()` to cast between time granules, which is efficient for regular time granules.

Usage

```
chronon_divmod(from, to, ...)
```

Arguments

`from`	The time granule that `x` is measured in (e.g., `day(1L)`).
`to`	The time granule to convert `x` into (e.g., `week(1L)`).
`...`	Additional arguments for methods.

Value

An list of two elements:

- `div`: integer vector of chronons measured in the `to` time granule.
- `mod`: integer vector of the remainder (in `from` time granule) after converting to the `to` time granule.

Examples

```
# Convert day 16 after epoch (1970-01-01) into weeks since epoch (and remainder days)
with(cal_isoweek, chronon_divmod(day(1L), week(1L), 16L))

# Convert week 4 after epoch (1970-W1) into days since epoch
with(cal_isoweek, chronon_divmod(week(1L), day(1L), 4L))
```

`chronon_epoch` *Epoch offset for chronons*

Description

Returns the epoch offset for a given chronon (time unit). The epoch defines the starting point of the chronon's linear numbering, used when converting from internal representations to common displays (e.g. applying an epoch of 1970-01-01).

Usage

```
chronon_epoch(x, ...)
```

Arguments

`x`	A chronon (time unit) object.
`...`	Additional arguments for methods.

Value

A numeric value representing the epoch for the chronon.

Examples

```
# The epoch for year linear time displays is 1970
chronon_epoch(cal_gregorian$year(1L))
```

`chronon_format_attr` *Default formatting strings for chronon attributes*

Description

Provides suffixes for default formatting strings for a given chronon (time granule). This provides useful information such as timezones or locations in the string.

Usage

```
chronon_format_attr(x, ...)
```

Arguments

x	A chronon (time granule) object.
...	Additional arguments for methods.

Value

A character string containing the default format suffix for the chronon.

---

chronon_format_linear *Default formatting strings for chronons*

Description

Provides default linear time formatting strings for a given chronon (finest time granule). The format strings use placeholders like `{lin(year(1L))}`, `{cyc(month(1L), year(1L)}` and `{cyc(day(1L), month(1L)}`, which are evaluated in the context of the data's `time_calendar()`.

Usage

```
chronon_format_linear(x, cal = time_calendar(x), ...)

chronon_format_cyclical(x, y, ...)

chronon_format_duration(x, ...)
```

Arguments

x	A time granule for the chronon.
cal	The calendar of the chronon, used to disambiguate suitable format strings for time units that are shared across calendars (e.g. `cal_gregorian$day` and `cal_isoweek$day`).
...	Additional arguments for methods.
y	A time granule for the cycle

Value

A character string containing the default format template for the chronon.

Examples

```
chronon_format_linear(cal_gregorian$year(1L))
chronon_format_linear(cal_gregorian$month(1L))
chronon_format_linear(cal_gregorian$day(1L))
chronon_format_linear(cal_isoweek$day(1L))

chronon_format_cyclical(cal_gregorian$month(1L), cal_gregorian$year(1L))
chronon_format_cyclical(cal_gregorian$day(1L), cal_gregorian$month(1L))
chronon_format_cyclical(cal_isoweek$day(1L), cal_isoweek$week(1L))
```

```
chronon_format_cyclical(cal_isoweek$week(1L), cal_isoweek$year(1L))

chronon_format_duration(cal_gregorian$year(1L))
chronon_format_duration(cal_gregorian$month(1L))
chronon_format_duration(cal_gregorian$day(1L))
```

`chronon_glb` *Find the finest or coarsest common chronon of a time object*

Description

These utility functions take a set of chronons and identify a common chronon that can represent all input chronons without loss of information, using the ordered relationships defined by `chronon_cardinality()` methods. This is useful for operations that require a shared time granule, such as combining or comparing different time measured at different precisions.

`chronon_glb()` finds the greatest lower bound (GLB): the common chronon of the *finest* granularity that can represent all input chronons without loss of information.

`chronon_lub()` finds the least upper bound (LUB): the common chronon of the *coarsest* granularity that every input chronon evenly aggregates into - the dual of `chronon_glb()`.

[Deprecated]

`chronon_common()` was renamed to `chronon_glb()` to distinguish it from its dual, `chronon_lub()`.

Usage

```
chronon_glb(x, ...)

chronon_glb.mixtime(x, .ptype = NULL, ...)

chronon_lub(x, ...)

chronon_lub.mixtime(x, .ptype = NULL, ...)

chronon_common(x, .ptype = NULL, ...)
```

Arguments

`x`	A time object (typically a `mixtime`).
`...`	Additional arguments for methods.
`.ptype`	If NULL, the default, the output returns the common chronon across all chronons of `x`. Alternatively, a prototype chronon can be supplied to `.ptype` to demand a specific chronon is used. If the supplied `.ptype` cannot represent all input chronons without loss of information, an error is raised.

Value

A time granule object representing the common chronon.

Examples

```
# The finest common chronon between a year-month and a day is a day
chronon_glb(c(yearmonth(Sys.Date()), date(Sys.Date())))

# The finest common chronon between a Gregorian month and an ISO week is a day
chronon_glb(c(yearweek(Sys.Date()), yearmonth(Sys.Date())))

# The finest common chronon between a ISO week and an hour is an hour
chronon_glb(c(yearweek(Sys.Date()), linear_time(Sys.time(), hour(1L))))

# The coarsest common chronon between a second and a minute is a minute
chronon_lub(c(linear_time(Sys.time(), second(1L)), linear_time(Sys.time(), minute(1L))))
```

---

chronon_parse_linear *Default parsing format strings for chronons*

Description

Provides candidate format strings for parsing text into a given chronon (finest time granule), for use as the `format` argument of `time_parse()`. Dispatches the same way as `chronon_format_linear()`/`chronon_format_cycl` but returns every common format instead of a single default, so `time_parse()` can try each in turn and keep whichever parses the most values. Methods should build their return value with `parse_format()`.

Usage

```
chronon_parse_linear(x, cal = time_calendar(x), ...)

chronon_parse_cyclical(x, y, ...)
```

Arguments

`x`	A time granule for the chronon.
`cal`	The calendar of the chronon, used to disambiguate suitable format strings for time units that are shared across calendars (e.g. `cal_gregorian$day` and `cal_isoweek$day`).
`...`	Additional arguments for methods.
`y`	A time granule for the cycle

Value

A character vector of format templates, ordered from most to least common, built with `parse_format()`. The first element typically matches `chronon_format_linear()` (for `chronon_parse_linear()`) or `chronon_format_cyclical()` (for `chronon_parse_cyclical()`) for the same chronon.

See Also

parse_format() for building candidate format strings (what these methods are built from), time_parse() for using the result as candidate formats, chronon_format_linear()/chronon_format_cyclical() for the single default format these are based on.

Examples

```
chronon_parse_linear(cal_gregorian$year(1L))
chronon_parse_linear(cal_gregorian$month(1L))
chronon_parse_linear(cal_gregorian$day(1L))
chronon_parse_linear(cal_isoweek$day(1L))

chronon_parse_cyclical(cal_gregorian$month(1L), cal_gregorian$year(1L))
chronon_parse_cyclical(cal_isoweek$day(1L), cal_isoweek$week(1L))
```

---

circsum *Compute circular rolling sums*

Description

Calculates rolling sums of length k for all contiguous subsequences around a circular vector. Returns sums for each valid k-element window that wraps around the vector as if arranged in a circle.

Usage

```
circsum(x, size, step = size)
```

Arguments

x	A numeric vector to compute circular sums over.
size	Integer; the window size (number of consecutive elements to sum). A negative size anchors the window at its end, with the window counting backwards.
step	Integer; the step size (the increment in starting index for each sum). A negative step walks the windows backwards around the circle.

Value

A numeric vector containing the sum of each contiguous subsequence around the circle. The length of the resulting vector is the number of combinations until the pattern between x and step repeats

Examples

```
# Simple circular sum with window of 2
circsum(c(1, 2, 3, 4), 2)
# Returns: 3 7 (1+2, 3+4)

# Window of 3 elements
circsum(c(1, 2, 3, 4, 5), 3)
# Returns: 6 10 9 8 12 (1+2+3, 4+5+1, 2+3+4, 5+1+2, 3+4+5)

# Negative step walks the same windows backwards
circsum(c(1, 2, 3, 4, 5), 1, -1)
# Returns: 1 5 4 3 2

# Negative size anchors each window at its end instead of its start - a
# trailing rather than leading rolling sum
circsum(c(1, 2, 3, 4, 5), -2, 1)
# Returns: 6 3 5 7 9 (5+1, 1+2, 2+3, 3+4, 4+5)
```

---

class_mixtime *Base S7 class for mixtime vector objects*

Description

`class_mixtime` is the base S7 class for all mixtime vector objects, inheriting from vecvec::class_vecvec. While not intended to be used directly, this S7 class is suitable to use when defining S7 methods for mixtime vectors. S3 methods can be defined using the `mixtime::mixtime` class.

Usage

```
class_mixtime(x = list(), i = seq_len(sum(lengths(x))))
```

Arguments

`x`	A list of `"mt_time"` vectors, see `mt_linear()` for details.
`i`	A vector of integers specifying the location of each element in `x` as if they were combined in order. The values in `i` must be between 1 and the total number of elements across all vectors in `x`, and can contain duplicates. If not provided, it defaults to a sequence from 1 to the total number of elements across all vectors in `x`.

Value

When used as a class definition (e.g., in `S7::method(generic, class_mixtime)`), an S7 class object representing the `mixtime` class, inheriting from vecvec::class_vecvec. When called as a constructor (`class_mixtime(list(...))`), a mixtime vector of S7 class `mixtime` (also inheriting the S3 class `"mixtime"`), containing the supplied list of time vectors as a vecvec::class_vecvec structure. End users should prefer `mixtime()` or `new_mixtime()` for construction.

See Also

`mixtime()` for creating mixtime vectors, and `new_mixtime()` for the low-level constructor function of this S7 class.

`component_helpers` *Linear and cyclical component helpers*

Description

`lin()` and `cyc()` name the time components addressed by mixtime's *component-aware contexts* — a single vocabulary shared across `format()` (and parsing) format strings and `time_components()` expressions. They are only meaningful inside one of these contexts; calling them directly is an error.

Usage

```
lin(granule)

cyc(granule, cycle)
```

Arguments

`granule`	The time granule to address, given as a granule generator (e.g. `year`) or a sized time unit (e.g. `year(1L)`). Resolved in the calendar of the time vector being formatted or decomposed.
`cycle`	The coarser granule defining the cycle a `cyc()` component repeats within (e.g. `year` in `cyc(month, year)`).

Details

A linear and a cyclical component of the same granule store the *same* value: the count of that chronon since the Unix epoch. The reduction to a within-cycle position happens only when a cyclical vector is formatted. `cyc()` therefore behaves like `lin()` on its finest granule but additionally records the cycle.

Value

A component specification, consumed internally by the component-aware context (e.g. `format()` or `time_components()`).

See Also

`format()` and `time_components()`

`cyclical_time` *Cyclical time points*

Description

`cyclical_time()` creates a vector of cyclical time points representing positions within repeating cycles. This function is useful for creating custom cyclical time representations that aren't covered by the convenience functions like `day_of_week()` or `month_of_year()`.

Usage

```
cyclical_time(
  data,
  chronon = time_chronon(data),
  cycle = time_cycle(data),
  discrete = TRUE,
  calendar = time_calendar(data)
)
```

Arguments

`data`	Input data to convert to cyclical time. Can be: • Numeric values (interpreted as chronons, 1-indexed) • Character strings (parsed as dates/times) • Date or POSIXct objects • Other time objects
`chronon`	A time granule representing the chronon (finest indivisible time granule), evaluated in the context of `calendar`. Use unquoted expressions like `day(1L)` or `month(1L)`. Chronons from a specific calendar can also be used (e.g. `cal_isoweek$day(1L)`).
`cycle`	A time granule representing the cycle (coarser time granule that defines the period), evaluated in the context of `calendar`. Use unquoted expressions like `week(1L)` or `year(1L)`.
`discrete`	Logical. If `TRUE` (default), returns integer positions within the cycle (discrete time model). If `FALSE`, returns fractional positions allowing representation of fractional time chronons (continuous time model).
`calendar`	Calendar system used to evaluate `chronon` and `cycle`. Defaults to `time_calendar(data)` for existing time objects. Common options include cal_gregorian and cal_isoweek.

Value

A `mixtime` time vector containing an `mt_cyclical` vector.

See Also

- new_cyclical_time_fn() for creating reusable cyclical time functions
- day_of_week(), day_of_month(), day_of_year() for common cyclical representations
- month_of_year(), week_of_year() for other cyclical time helpers
- cal_gregorian, cal_isoweek for calendar systems

Examples

```
# Day of week (1-7, Monday = 1)
cyclical_time(
  Sys.Date(),
  chronon = day(1L),
  cycle = week(1L),
  calendar = cal_isoweek
)

# Month of year (1-12)
cyclical_time(
  Sys.Date(),
  chronon = month(1L),
  cycle = year(1L)
)

# Continuous time (discrete = FALSE) for fractional month of year
cyclical_time(
  Sys.Date(),
  chronon = month(1L),
  cycle = year(1L),
  discrete = FALSE
)

# Day of month with Gregorian calendar
cyclical_time(
  Sys.Date(),
  chronon = day(1L),
  cycle = month(1L),
  calendar = cal_gregorian
)

# Hours, minutes, and seconds
cyclical_time(
  Sys.time(),
  chronon = second(1L),
  cycle = day(1L)
)
```

cyclical_time_helpers *Cyclical time helpers*

Description

Helper functions for creating cyclical time representations. These functions create time objects that repeat within a larger time cycle, useful for identifying seasonal patterns or positions within a calendar period.

Usage

```
month_of_year(data, discrete = TRUE, calendar = time_calendar(data), ...)

day_of_year(data, discrete = TRUE, calendar = time_calendar(data), ...)

day_of_month(data, discrete = TRUE, calendar = time_calendar(data), ...)

time_of_day(data, discrete = TRUE, calendar = time_calendar(data), ...)

day_of_week(data, discrete = TRUE, calendar = time_calendar(data), ...)

week_of_year(data, discrete = TRUE, calendar = time_calendar(data), ...)
```

Arguments

data	Another object to be coerced into the specified cyclical time.
discrete	If TRUE, the position within the cycle that data falls into is returned as an integer. If FALSE, a fractional position is returned (analagous to time using a continuous time model).
calendar	A calendar object specifying the calendar system to use.
...	Additional arguments for cyclical_time(), such as tz for timezones.

Value

A mixtime time vector containing an mt_cyclical vector with chronon and cycle matching the function used.

Cyclical time representations

- day_of_week(): Represents the day position within a week (1-7) using the ISO 8601 standard where weeks start on Monday.
- day_of_month(): Represents the day position within a month (1-28, 1-29, 1-30, or 1-31 depending on the month). The chronon is one day, cycling within a month.
- day_of_year(): Represents the day position within a year (1-365 or 1-366 for leap years). The chronon is one day, cycling within a year.

- `week_of_year()`: Represents the week position within a year (1-52 or 1-53) using the ISO 8601 week numbering system.
- `month_of_year()`: Represents the month position within a year (1-12). The chronon is one month, cycling within a year.

Custom cyclical time representations

You can create custom cyclical time representations using `cyclical_time()` with any of the supported time units (see calendar_gregorian and calendar_isoweek).

For example, to create a representation for day of the month:

```
day_of_month <- new_cyclical_time_fn(
  chronon = day(1L), cycle = month(1L),
  default_calendar = cal_gregorian
)
```

See Also

`cyclical_time()` for creating cyclical time vectors, `new_cyclical_time_fn()` for creating cyclical time helper functions

Examples

```
month_of_year(Sys.Date())
day_of_year(Sys.Date())
day_of_week(Sys.Date())
day_of_week(as.Date("2025-12-15") + 0:6)
```

---

duration *Duration vectors*

Description

`duration()` creates a vector of durations with a specified chronon. Durations represent a fixed span of time measured in a given time granule (e.g., 3 months, 5 days), without reference to a specific point in time.

Usage

```
duration(
  data,
  chronon = time_chronon(data),
  discrete = NULL,
  calendar = time_calendar(data)
)
```

Arguments

data	A time vector of duration magnitudes, or an existing duration() vector to convert to chronon granules.
chronon	A time granule expression representing the chronon, evaluated in the context of calendar. Use unquoted expressions like month(1L) or day(1L). Chronons from a specific calendar can also be used (e.g. cal_gregorian$month(1L)). Defaults to the time chronon of the input data (time_chronon(data)).
discrete	Logical. If TRUE (default), returns integer durations counting whole granules, rounding towards zero (discrete time model). If FALSE, returns fractional durations (continuous time model).
calendar	Calendar system used to evaluate chronon. Defaults to time_calendar(data) for existing time objects. Common options include cal_gregorian and cal_isoweek.

Value

A mixtime vector containing an mt_duration vector.

See Also

- new_duration_fn() for creating reusable duration functions
- cal_gregorian, cal_isoweek for calendar systems

Examples

```
# A duration of 3 months
duration(3L, cal_gregorian$month(1L))

# A vector of durations in days
duration(1:7, cal_gregorian$day(1L))

# Convert a duration of 4 days into weeks
duration(days(4), cal_isoweek$week(1L), discrete = FALSE)
duration(days(4), cal_isoweek$week(1L), discrete = TRUE)
```

---

duration_helpers *Duration helper functions*

Description

Convenience functions for creating duration vectors of common time units. Each function wraps new_duration_fn() for its respective chronon.

Usage

```
years(data, calendar = time_calendar(data), ...)

quarters(data, calendar = time_calendar(data), ...)

months(data, calendar = time_calendar(data), ...)

weeks(data, calendar = time_calendar(data), ...)

days(data, calendar = time_calendar(data), ...)

hours(data, calendar = time_calendar(data), ...)

minutes(data, calendar = time_calendar(data), ...)

seconds(data, calendar = time_calendar(data), ...)

milliseconds(data, calendar = time_calendar(data), ...)
```

Arguments

data	A time vector of duration magnitudes, or an existing duration() vector to convert to chronon granules.
calendar	Calendar system used to evaluate chronon. Defaults to time_calendar(data) for existing time objects. Common options include cal_gregorian and cal_isoweek.
...	Additional arguments passed to the chronon (e.g. tz for timezones).

Value

A mixtime vector containing an mt_duration vector.

See Also

- new_duration_fn() for creating custom duration functions
- duration() for creating duration vectors directly
- cal_gregorian, cal_isoweek for calendar systems

Examples

```
years(3L)
quarters(2L)
months(6L)
weeks(4L)
days(7L)
hours(12L)
minutes(30L)
seconds(45L)
milliseconds(500L)
```

format.mt_time *Format mixtime vectors*

Description

Formats a mixtime vector as a character vector, using a glue-style format string of `{lin(...)}`/`{cyc(...)}` tokens tailored to the vector's chronon and cycle. If `format` is omitted, a sensible default is derived automatically. See `vignette("time-format-strings")` for the format string syntax.

Usage

```
## S4 method for signature 'mt_time'
format(x, ..., attr = TRUE)
```

Arguments

`x`	A mixtime vector.
`...`	Additional arguments for methods, including `format`: a glue-style format string, defaulting to one derived automatically for `x`.
`attr`	If `TRUE` (default), append attribute information (e.g. timezone) to the default format.

Value

A character vector the same length as `x`.

is_mixtime *Check if an object is a mixtime*

Description

Tests whether `x` inherits from the `mixtime` class.

Usage

```
is_mixtime(x)
```

Arguments

`x`	An object to test.

Value

A scalar logical: `TRUE` if `x` is a `mixtime` vector, `FALSE` otherwise.

See Also

`as_mixtime()` to coerce objects to `mixtime`, `mixtime()` to construct a `mixtime`.

Examples

```
is_mixtime(Sys.Date())
is_mixtime(mixtime(Sys.Date()))
```

---

is_time *Check the time type of values*

Description

Test whether elements of a `mixtime` vector are linear, cyclical, or durations.

Usage

```
time_is_linear(x, ...)

time_is_cyclical(x, ...)

time_is_duration(x, ...)

is_time_linear(x, ...)

is_time_cyclical(x, ...)

is_time_duration(x, ...)
```

Arguments

`x`	A time object (typically a `mixtime` vector).
`...`	Additional arguments for methods.

Details

These helpers return a logical vector the same length as `x` identifying the type of time represented by each element.

Value

A logical vector the same length as `x`.

Examples

```
t <- c(yearmonth(0), month_of_year(0), months(0L))
time_is_linear(t)
time_is_cyclical(t)
time_is_duration(t)
```

`label_scheme` *Describe a granule's label scheme*

Description

Builds the function a `linear_labels()`/`cyclical_labels()` method is: assign its result directly, e.g. `method(cyclical_labels, list(granule, cycle)) <- label_scheme(...)`. Three levels, from plain to most irregular:

`linear_labels()` and `cyclical_labels()` are S7 generics returning a granule's label scheme: a plain list describing how its internal position renders as text and parses back (`start`, `vocab`, `transform`, `width`, `locale`; see `label_scheme()`).

Usage

```
label_scheme(
  start = 0L,
  vocab = NULL,
  transform = NULL,
  width = NULL,
  locale = NULL
)

linear_labels(granule, ...)

cyclical_labels(granule, cycle, ...)
```

Arguments

`start`	Author-fixed. Raw index 0 displays as `start`, shared by numeric rendering and vocab indexing (e.g. `start = 1L` makes January, raw index 0, display as `"1"` and index `vocab()[[1]]`). A caller can't override this: get it wrong and dates are wrong, not just styled differently.
`vocab`	`NULL` (numeric only), or a function `function(type = NULL, locale = NULL)` returning either the full named list of renderings for one locale (`type = NULL`), or one rendering (e.g. `vocab("abbreviated")`). `vocab_table()` builds this from a plain lookup table; see "Locale specific labels" below for how to specify labels.

`transform`	`NULL` (plain `start` shift), or `list(encode = function(i, at = NULL) -> vocab index, decode = func` overriding the indexing used for named rendering only. Both functions are authored scalar (`if`/`return`) and vectorized internally. See "Irregular cycles" below.
`width`, `locale`	Caller-overridable defaults for `cyc(granule, cycle, width = ...)`/`locale = ...` at format/parse time, not calendar facts. `width` zero-pads numeric rendering (e.g. `width = 2L` for `"02"`); `locale` selects which `vocab()` entry to use by default.
`granule`	A time granule instance (e.g. `month(1L)`).
`...`	Unused; only present for S7 dispatch.
`cycle`	For `cyclical_labels()`, the coarser granule instance this cycle repeats within (e.g. `year(1L)`).

Details

1. Plain numeric. No `vocab`: labels are `i + start`, optionally zero-padded to `width`. The default for most granules (minutes, seconds).
2. Named, regular. `vocab` supplies the names (see `vocab_table()`); `start` positions raw index 0 in both the numeric and vocab-indexed rendering. Covers most named calendar units (months, weekdays).
3. Named, irregular. `transform` overrides the plain `start` shift with hand-written encode/decode, for cycles where raw index and name aren't a constant offset apart (e.g. a leap month splitting one name into two). See "Irregular cycles" below.

A scheme is only consulted for named (`label = TRUE`) rendering and its inverse; numeric rendering is always the plain `i + start` shift (see `linear_labels_format()`). Labels that aren't index-shaped at all (e.g. "1BC") override the generics directly, documented in `linear_labels()` instead.

`linear_labels_format()`/`linear_labels_parse()` (and their cyclical counterparts) do the actual rendering/parsing, with the default method looking up the scheme and applying it. A granule with labels that are incompatible with label schemes (e.g. 2BC, 1BC, 1, 2, ...) registers a method directly on those generics instead, skipping the scheme system. See "Overriding the generics directly" below.

Value

A function `function(granule, cycle) list(start =, vocab =, transform =, width =, locale =)`, assignable directly as a `linear_labels()`/`cyclical_labels()` method. `granule`/`cycle` are only used for dispatch: a scheme describes a granule class, not the instance a method happens to be called with.

A plain list, as constructed by `label_scheme()`.

Locale specific labels

`vocab` is a plain closure, not a shared registry, so two calendar packages can't collide on it. To add locale support to an existing granule, redefine the `linear_labels()`/`cyclical_labels()` method and call the previous one for anything you're not changing: the normal S7 rule that the last `method(...) <-` wins. That needs a hand-written wrapper rather than assigning `label_scheme()`'s result directly, since overriding one field means fetching the old scheme and changing it:

```
scheme <- cyclical_labels(cal_gregorian$month(1L), cal_gregorian$year(1L))
method(cyclical_labels, list(cal_gregorian$month, cal_gregorian$year)) <-
  function(granule, cycle) {
    scheme$vocab <- function(type = NULL, locale = NULL) {
      if (identical(locale %||% "en-GB", "fr-FR")) {
        fr <- list(
          wide = c("janvier", "février", "mars", "avril", "mai", "juin",
              "juillet", "août", "septembre", "octobre", "novembre", "décembre"),
          abbreviated = c("janv.", "févr.", "mars", "avr.", "mai", "juin",
                          "juil.", "août", "sept.", "oct.", "nov.", "déc.")
        )
        return(if (is.null(type)) fr else fr[[type]])
      }
      scheme$vocab(type, locale)
    }
    scheme
  }
```

Locale tags follow BCP 47 (`"en-GB"`, `"fr-FR"`), like ICU, `stringi` and `clock`. Type names follow CLDR's wide/abbreviated/narrow, so CLDR data (e.g. `stringi::stri_datetime_symbols()`) works with `vocab_table()` unchanged. Other `type` values are allowed too, e.g. `"emoji"` for lunar phases, picked at render time via `cyclical_labels_format()`'s `type` argument.

Irregular cycles

`at` (in `transform`'s encode/decode, and in `cyclical_labels_format()`/`cyclical_labels_parse()`) is the coarser granule instance's raw position, not epoch-shifted, with `n = 1`. It follows the same convention as `chronon_cardinality()`. It lets `transform` pick the right name when that name depends on which cycle instance a raw index falls in, e.g. the Hebrew calendar's leap year splitting "Adar" into "Adar I"/"Adar II":

```
month_names <- c("Tishrei", "Cheshvan", "Kislev", "Tevet", "Shevat",
                 "Adar", "Adar I", "Adar II",
                 "Nisan", "Iyar", "Sivan", "Tammuz", "Av", "Elul")

method(cyclical_labels, list(cal_hebrew$month, cal_hebrew$year)) <- label_scheme(
  start = 1L,
 vocab = vocab_table(`en-GB` = list(wide = month_names, abbreviated = month_names)),
  transform = list(
    encode = function(i, at) {
      leap <- is_hebrew_leap_year(at)
      if (i <= 4L) return(i + 1L)
      if (!leap) return(if (i == 5L) 6L else i + 3L)
      if (i == 5L) return(7L)
      if (i == 6L) return(8L)
      i + 2L
    },
    decode = function(d, at) {
      leap <- is_hebrew_leap_year(at)
```

```
      if (d <= 5L) return(d - 1L)
      if (!leap) {
        if (d == 6L) return(5L)
      if (d %in% c(7L, 8L)) cli::cli_abort("'Adar I'/'Adar II' are not valid outside a leap year.")
        return(d - 3L)
      }
    if (d == 6L) cli::cli_abort("'Adar' is ambiguous in a leap year, use 'Adar I' or 'Adar II'.")
      if (d == 7L) return(5L)
      if (d == 8L) return(6L)
      d - 2L
    }
  )
)
```

The aborts in `decode()` matter: without them, "Adar II" typed in a common year would fall through to the same offset as plain "Adar" and silently parse to the wrong month. Only the label layer, which knows both the text and `at`, can catch that. `transform`'s leap predicate must also be the same one `chronon_cardinality()` uses for this pair; put it in one shared helper.

Overriding the format and parsing methods

`linear_labels_format()`/`linear_labels_parse()` are ordinary S7 generics with a default method, not the only place formatting logic can live. So escaping the scheme system for one granule is just registering a method directly:

```
method(linear_labels_format, cal_gregorian$year) <- function(granule, i, ...) {
  ifelse(i <= 0L, paste0(-i + 1L, "BC"), i)
}
method(linear_labels_parse, cal_gregorian$year) <- function(granule, ...) {
  list(
    pattern = "\\d+(?:BC)?",
    decode = function(text, at = NULL) {
      bc <- grepl("BC$", text)
      n <- as.integer(sub("BC$", "", text))
      ifelse(bc, 1L - n, n) - chronon_epoch(granule)
    }
  )
}
```

`decode()` needs `- chronon_epoch(granule)` because this method bypasses the default, which normally does that step (see `linear_labels_format()`'s "Epoch shift" section).

The `...` on `_format()` is not just style: `format()` forwards whatever component_helpers' `lin()`/`cyc()` were called with (`label`, `abbreviate`, `type`, `width`, `locale`, ...) as named arguments. A method without `...` errors with "unused argument" as soon as a caller passes something it doesn't declare.

Registering only `linear_labels_format()` is fine: that's the same "no parsing yet" state a fresh granule starts in. But registering it without `linear_labels_parse()` leaves parsing silently wrong instead of just missing: text like `"1BC"` falls through to whatever the granule's scheme says, usually plain numeric decoding, which doesn't understand "BC" at all.

See Also

linear_labels()/cyclical_labels(), the generics a scheme is returned from; vocab_table() for the common hand-listed name table case.

label_scheme() for the fields a method returns; linear_labels_format()/cyclical_labels_format() for the generics that interpret it; vocab_table() for the common hand-listed name table case.

Examples

```
# A scheme for months of the year, using R's localised month names.
S7::method(cyclical_labels, list(cal_gregorian$month, cal_gregorian$year)) <- label_scheme(
  start = 1L, width = 2L,
  vocab = vocab_table(`en-GB` = list(wide = month.name, abbreviated = month.abb))
)
```

---

linear_labels_format *Render and parse a granule's labels*

Description

The generics turning a granule's internal position into display text (linear_labels_format() for a non-repeating position, e.g. the year; cyclical_labels_format() for a position within a larger cycle, e.g. the month within the year) and back (linear_labels_parse()/ cyclical_labels_parse()).

Usage

```
linear_labels_format(granule, ...)

cyclical_labels_format(granule, cycle, ...)

linear_labels_parse(granule, ...)

cyclical_labels_parse(granule, cycle, ...)
```

Arguments

granule	A time granule object representing the granule (e.g. month(1L)).
...	Passed on to the method. The default method (see below) takes: • i: Integer vector: the position along the linear axis, or within the cycle for cyclical_labels_format(). • at: The linear position of the cycle granule, letting a method produce labels specific to that cycle instance (mainly for irregular cycles, e.g. a leap month). Same convention as at in chronon_cardinality(). • label: If TRUE, return named labels (e.g. "February"). If FALSE, return the numeric position as character.

- `abbreviate`: If `TRUE`, return abbreviated labels (e.g. "Feb"). If `FALSE`, return full labels (e.g. "February").
- `type`: Overrides `abbreviate`'s abbreviated/wide choice of `vocab_table()` type outright, e.g. `type = "emoji"`. `NULL` (default) defers to `abbreviate`.
- `width`,`locale`: Call-time overrides of the scheme's `width`/`locale` defaults (see `label_scheme()`); `NULL` defers to the scheme.

`cycle` A time granule object representing the cycle (e.g. `year(1L)`).

Details

Each has one default method (on the base `mt_unit` class) that looks up the granule's scheme via `linear_labels()`/`cyclical_labels()` and applies it (see `label_scheme()` for the fields). Calendar authors usually declare a scheme instead of writing a method here directly. Writing one directly overrides the generics for labels that aren't index-shaped at all, see `linear_labels()`'s "Overriding the generics directly" section.

Value

Character vector of labels for the time point.

Epoch shift

`linear_labels_format()`'s `i` arrives already epoch-shifted for display (`chronon_parts()` adds `chronon_epoch()` before calling it, e.g. a Gregorian year's `i` is `2024`, not `54`). The default `linear_labels_parse()` method shifts `decode()`'s output back, by subtracting `chronon_epoch()` after the scheme's own decode runs. A hand-written override method skips this and must do the same subtraction itself if the granule has a non-zero epoch; see the example in `linear_labels()`.

See Also

`linear_labels()`/`cyclical_labels()` for the authoring interface these generics' default methods interpret.

Examples

```
# Labels for years on a linear axis
with(cal_gregorian, linear_labels_format(year(1L), 2020:2025))

# Labels for months in a year
with(cal_gregorian, cyclical_labels_format(month(1L), year(1L), 0:11))
```

`linear_time` *Linear time points*

Description

`linear_time()` creates a vector of linear time points with a specified chronon (smallest time granule). This function is useful for creating custom time representations that aren't covered by the convenience functions like `yearmonth()` or `yearweek()`.

Usage

```
linear_time(
  data,
  chronon = time_chronon(data),
  discrete = TRUE,
  calendar = time_calendar(data)
)
```

Arguments

`data`	Input data to convert to linear time. Can be: • Numeric values (interpreted as chronons since Unix epoch) • Character strings (parsed as dates/times) • Date or POSIXct objects • Other time objects
`chronon`	A time granule expression representing the chronon (smallest indivisible time granule), evaluated in the context of `calendar`. Use unquoted expressions like `month(1L)` or `hour(1L)`. Chronons from a specific calendar can also be used (e.g. `cal_isoweek$week(1L)`). Defaults to the time chronon of the input `data` (`time_chronon(data)`).
`discrete`	Logical. If `TRUE` (default), returns integer chronons since Unix epoch (discrete time model). If `FALSE`, returns fractional chronons allowing representation of fractional time granules (continuous time model).
`calendar`	Calendar system used to evaluate `chronon` and `granules`. Defaults to `time_calendar(data)` for existing time objects. Common options include cal_gregorian and cal_isoweek.

Value

A `mixtime` time vector containing an `mt_linear` vector.

See Also

- `new_linear_time_fn()` for creating reusable linear time functions
- `yearmonth()`, `yearquarter()`, `year()` for Gregorian time representations
- `yearweek()` for ISO 8601 week-based time
- cal_gregorian, cal_isoweek for calendar systems

Examples

```
# Hourly time
linear_time(
  Sys.time(),
  chronon = hour(1L)
)

# Monthly time
linear_time(
  Sys.Date(),
  chronon = month(1L)
)

# Discrete vs continuous time
linear_time(Sys.time(), chronon = day(1L), discrete = TRUE)
linear_time(Sys.time(), chronon = day(1L), discrete = FALSE)

# ISO week calendar with week-day structure
linear_time(
  Sys.Date(),
  chronon = day(1L),
  calendar = cal_isoweek
)
```

---

linear_time_helpers *Linear time helper functions*

Description

Convenience functions for creating common linear time representations. These functions work with different calendar systems and adapt based on the input data's calendar.

Usage

```
year(data, discrete = TRUE, calendar = time_calendar(data), ...)

yearquarter(data, discrete = TRUE, calendar = time_calendar(data), ...)

yearmonth(data, discrete = TRUE, calendar = time_calendar(data), ...)

yearweek(data, discrete = TRUE, calendar = time_calendar(data), ...)

date(data, discrete = TRUE, calendar = time_calendar(data), ...)

datetime(data, discrete = TRUE, calendar = time_calendar(data), ...)
```

Arguments

data	A vector of time points (e.g. base::Date, base::POSIXt)
discrete	If TRUE, the number of chronons since Unix epoch that data falls into is returned as an integer. If FALSE, a fractional number of chronons is returned (analagous to time using a continuous time model).
calendar	A calendar used to evaluate the time units. Defaults to the calendar of the input data. Common options include cal_gregorian and cal_isoweek.
...	Additional arguments for linear_time(), such as tz for timezones.

Details

These functions create linear time representations with different chronons and granules:

- year(): Represents time in whole years. The chronon is one year.
- yearquarter(): Represents time in quarters, grouped by year. The chronon is one quarter, with years as the granule.
- yearmonth(): Represents time in months, grouped by year. The chronon is one month, with years as the granule.
- yearweek(): Represents time in weeks, grouped by year. The chronon is one week, with years as the granule. Defaults to ISO 8601 week calendar.

Value

A mixtime time vector containing an mt_linear vector with chronons matching the function used.

Calendar flexibility

These functions adapt to the calendar system of the input data. For example:

- year(date("2025-12-29")) returns a Gregorian year
- year(yearweek(date("2025-12-29"))) returns an ISO week-based year

You can also explicitly specify a calendar using the calendar argument:

```
year(date("2025-12-29"), calendar = cal_isoweek)
```

Custom linear time representations

For more complex time structures, use linear_time() or new_linear_time_fn() to create custom representations with any combination of chronons and granules.

See Also

- linear_time() for creating custom linear time representations
- new_linear_time_fn() for creating reusable linear time functions
- cal_gregorian, cal_isoweek for calendar systems

Examples

```
# Gregorian year
year(Sys.Date())
year(Sys.Date(), discrete = FALSE)

# ISO week-based year
year(yearweek(Sys.Date()))

# Year-quarter
yearquarter(Sys.Date())
yearquarter(Sys.Date(), discrete = FALSE)

# Year-month
yearmonth(Sys.Date())
yearmonth(Sys.Date(), discrete = FALSE)

# Year-week (ISO 8601)
yearweek(Sys.Date())
yearweek(0:52)
```

---

mixtime *Create a mixtime vector*

Description

A mixtime is a vector which describes a point in time. It uses a calendar definition to translate a vector of numbers into a point in time.

Usage

```
mixtime(
  data,
  chronon = time_chronon(data),
  cycle = time_cycle(data),
  discrete = TRUE
)
```

Arguments

data	A vector of time values. This can be a character vector (e.g. "2024-01-01"), a numeric vector (e.g. seconds since epoch), or a time class (e.g. Date, POSIXct, yearmonth, etc.).
chronon	A time granule object representing the smallest indivisible time granule (chronon) for the mixtime. This is used to interpret the numeric values in data and to define the time resolution of the mixtime. If not provided, it will be inferred from data.

`cycle` An optional time granule object representing the cycle for cyclical time. This is used to define the repeating cycle for cyclical time representations (e.g. day-of-week, month-of-year). If not provided, the mixtime will be treated as linear time.

`discrete` A logical indicating whether the time values should be treated as discrete (integer) or continuous (fractional). This affects how numeric values are interpreted and how time arithmetic is performed. The default is `TRUE` (discrete).

Value

A `mixtime` object representing the time values in `data` according to the specified `chronon` and `cycle`.

Examples

```
# Create a mixtime for today
mixtime(Sys.Date())

# Create a mixtime for the current date and time
mixtime(Sys.time())

# Convert time from tsibble classes to mixtime
mixtime(tsibble::yearmonth("2024 Jan"))

# Create a mixtime for the time of day (cyclical time)
mixtime(Sys.time(), cycle = cal_gregorian$day(1L))

# Specify a timezone for the chronon
mixtime(Sys.time(), chronon = cal_gregorian$second(1L, tz = Sys.timezone()))
mixtime(Sys.time(), chronon = cal_gregorian$second(1L, tz = "Pacific/Honolulu"))
mixtime(Sys.time(), chronon = cal_gregorian$second(1L, tz = "Australia/Melbourne"))

# Dates (and all granularities) can have timezones
mixtime(Sys.time(), chronon = cal_gregorian$day(1L, tz = Sys.timezone()))
mixtime(Sys.time(), chronon = cal_gregorian$day(1L, tz = "Pacific/Honolulu"))
mixtime(Sys.time(), chronon = cal_gregorian$day(1L, tz = "Australia/Melbourne"))

# Continuous time tracks progress within the chronon
mixtime(Sys.time(), chronon = cal_gregorian$day(1L, tz = Sys.timezone()), discrete = FALSE)

# Mixtime can combine different granularities and timezones in a vector
now <- Sys.time()
c(
  # Datetime (second chronon) in UTC
  mixtime(now),
  # Date (minute chronon) in local timezone
  mixtime(now, chronon = cal_gregorian$minute(1L, tz = Sys.timezone())),
  # Month (month chronon) in UTC
  mixtime(now, chronon = cal_gregorian$month(1L))
)
```

mixtime_location *Extract locations from an object*

Description

Generic function to extract the location from objects that have location information.

Usage

```
loc_latitude(x, ...)

loc_longitude(x, ...)

loc_altitude(x, ...)
```

Arguments

x	An object with location information.
...	Additional arguments passed to methods.

Value

A numeric value representing the location (e.g., longitude, latitude, etc.).

Examples

```
t <- linear_time(
  1:3,
  cal_time_solar$day(1L, lat = -37.8136, lon = 144.9631)
)

loc_longitude(t)
loc_latitude(t)
loc_altitude(t)
```

mt_cyclical-compare *Comparison operators for cyclical time (*mt_cyclical*)*

Description

A cyclical time value stores an absolute chronon count but *means* a position within its cycle (e.g. `day_of_week()` means a weekday, not a particular Wednesday). Comparison therefore reduces both operands to their position within the cycle - the same reduction `format()` displays - and compares those:

- `day_of_week(date("2020-01-15")) == day_of_week(date("2020-01-22"))` is `TRUE`, since both are a Wednesday.
- `day_of_year(date("2020-01-15")) == day_of_year(date("2021-01-15"))` is `TRUE`, since both are the 15th day of their year.

Both operands must share a `cycle`: a cycle is a modulus rather than a unit, so there is no meaningful common cycle between (say) a weekday and a day-of-year, and comparing them is an error. Differing *chronons* within a shared cycle are reconciled as they are for mt_linear: both positions are expressed in the finest common chronon, and a discrete value spans the closed interval of its chronon, so

- `a == b` iff `start(a) == start(b)` and `end(a) == end(b)`
- `a < b`/`a > b` iff a/b ends at or before the other begins (a gap, or just adjacency - see mt_linear-compare for why this includes adjacency)
- `a <= b` iff `end(a) <= end(b)`, `a >= b` iff `start(a) >= start(b)`

As for `mt_linear`, this is not a total order when chronons differ. Ordering follows the position within the cycle (so `Mon < Wed`); the cycle's wrap-around is not treated as circular.

Usage

```
## S4 method for signature 'mt_cyclical'
e1 == e2
## S4 method for signature 'mt_cyclical'
e1 != e2
## S4 method for signature 'mt_cyclical'
e1 < e2
## S4 method for signature 'mt_cyclical'
e1 <= e2
## S4 method for signature 'mt_cyclical'
e1 > e2
## S4 method for signature 'mt_cyclical'
e1 >= e2
```

Arguments

`e1`, `e2`	`mt_cyclical` vectors sharing a cycle (or values castable to one, such as plain numeric vectors in the other operand's chronon, read from its epoch as `mixtime()` reads them).

Value

A logical vector.

`mt_duration-compare` *Comparison operators for durations (*`mt_duration`*)*

Description

A duration is a scalar magnitude of time measured in a given chronon (e.g. `days(3)`), with no reference to a point in time. Comparing two durations is therefore a plain magnitude comparison, once both operands have been expressed in a common chronon:

- a == b iff the two magnitudes are equal in their common chronon
- a < b, a <= b, a > b, a >= b compare the magnitudes directly

Unlike `mt_linear` comparison there is no interval/span to consider, so this *is* a total order. When both operands already share a chronon the magnitudes are compared as-is; otherwise both are scaled to their finest common chronon (the same scaling used when combining durations arithmetically, see `duration_combine()`).

Usage

```
## S4 method for signature 'mt_duration'
e1 == e2
## S4 method for signature 'mt_duration'
e1 != e2
## S4 method for signature 'mt_duration'
e1 < e2
## S4 method for signature 'mt_duration'
e1 <= e2
## S4 method for signature 'mt_duration'
e1 > e2
## S4 method for signature 'mt_duration'
e1 >= e2
```

Arguments

`e1`, `e2`	`mt_duration` vectors (or values castable to one, such as plain numeric vectors interpreted in the other operand's chronon).

Value

A logical vector.

mt_linear-compare *Comparison operators for linear time (*mt_linear*)*

Description

Discrete linear time values represent a *closed interval* spanning their chronon (e.g. `year(2020)` spans every instant from the start of 2020 to the end of 2020), while continuous linear time values represent a single instant. Comparing two `mt_linear` vectors therefore compares the start/end instants of the (possibly zero-width) interval each value represents:

- a == b iff `start(a) == start(b)` and `end(a) == end(b)`
- a < b iff `end(a) <= start(b)` (i.e. before with or without a gap)
- a > b iff `start(a) >= end(b)` (i.e. after with or without a gap)
- a <= b iff `end(a) <= end(b)` (right-bound comparison)
- a >= b iff `start(a) >= start(b)` (left-bound comparison)

This is not a total order for comparisons between time points at different granularities: <=/>= are **not** shorthand for `(< or ==)/(> or ==)`. For example, `yearquarter("2020 Q3") <= year("2020")` is TRUE despite `yearquarter("2020 Q3") < year("2020")` and `yearquarter("2020 Q3") == year("2020")` being FALSE.

The inequality operators </> and <=/>= are useful conjugations of Allen's interval algebra for common data manipulation needs. The complete set of Allen's 13 base relations are documented in allen-interval-algebra.

Usage

```
## S4 method for signature 'mt_linear'
e1 == e2
## S4 method for signature 'mt_linear'
e1 != e2
## S4 method for signature 'mt_linear'
e1 < e2
## S4 method for signature 'mt_linear'
e1 <= e2
## S4 method for signature 'mt_linear'
e1 > e2
## S4 method for signature 'mt_linear'
e1 >= e2
```

Arguments

e1, e2	`mt_linear` vectors (or values castable to one, such as plain numeric vectors in the other operand's chronon, read from its epoch as `mixtime()` reads them, so `year(x) == 2026L` compares with 2026).

Value

A logical vector.

`mt_time-class` *Time vector classes*

Description

The `mt_time` family are the S7 vector classes that store time points as a numeric count of chronons. `mt_time` is the (internal) base class carrying the `chronon` property; the common modes of time are:

- `mt_linear` - linear time points, typically produced with `linear_time()`.
- `mt_cyclical` - cyclical time points with additional `cycle` granule, typically produced with `cyclical_time()`.
- `mt_duration` - time durations, typically produced with `duration()`.

The underlying data can be either **integer** (discrete time) or **double** (continuous time). The `chronon` (and, for `mt_cyclical`, the `cycle`) are time granules, the result from a mt_unit object.

Usage

```
mt_linear(.data = integer(), chronon = mt_unit(1L))

mt_duration(.data = integer(), chronon = mt_unit(1L))

mt_cyclical(.data = integer(), chronon = mt_unit(1L), cycle = mt_unit(1L))
```

Arguments

`.data`	A numeric vector of chronon counts (integer or double).
`chronon, cycle`	Time granules (mt_unit objects) giving the unit of the chronon counts and, for `mt_cyclical`, the length of the cycle.

Value

An S7 class object (used for method dispatch), or a time vector when called as a constructor.

See Also

`mt_linear()`, `mt_duration()`, and `mt_cyclical()` to construct time vectors, and mt_unit for the granule type stored in `chronon`/`cycle`.

`mt_unit` *Base S7 class for creating new time units*

Description

This class is the primitive class for time units, and should be extended from when creating new time units. A new class is typically created with S7 using: `S7::new_class("tu_***", parent = mt_tz_unit)`

Usage

```
mt_unit(n = 1L)

mt_loc_unit(n = 1L, lat = naive_loc, lon = naive_loc, alt = naive(0))

mt_tz_unit(n = 1L, tz = naive_tz)
```

Arguments

`n`	The step size of time granule. For example, `n = 2L` is 2 time units, and `cal_isoweek$week(2L)` would represent 2 weeks (a fortnight).
`lat`	Numeric. Latitude in decimal degrees. Range: -90 to 90. Default: naive location (astronomical calculations require `lat`).
`lon`	Numeric. Longitude in decimal degrees. Range: -180 to 180. Default: naive location (astronomical calculations require `lon`).
`alt`	Numeric. Altitude in meters above sea level. Default: 0 (sea level).
`tz`	The timezone name for the unit (valid units can be found with `[tzdb::tzdb_names()]`)

Details

Time units are the building blocks of calendars in mixtime. Each unit represents a specific temporal component (e.g., day, month, year) and can be combined using `new_calendar()` to create a calendar system.

When creating custom calendars, define time unit classes that inherit from either `mt_unit` (for standard units) or `mt_tz_unit` (for timezone-aware units), then pass them as named arguments to `new_calendar()`. The calendar will use these names to create constructor functions accessible via `$` notation (e.g., `calendar$day(1L)`).

Value

A time granule object of class `mt_unit`

Calendar Algebra Methods

Time units enable calendar arithmetic through key generic methods that should be implemented for custom time units:

- `chronon_cardinality_fixed(x, y)` - Returns the number of unit `x` granules that fit within one unit `y` granule, for relationships that are a constant, context-independent number (e.g., 7 days per week, 24 hours per day). Prefer this over `chronon_cardinality()` whenever the relationship does not depend on `at`, since it is also used to determine which relationships are safe to use for `chronon_divmod()`'s graph traversal.
- `chronon_cardinality(x, y, at)` - Returns the number of `x` granule that fit within one `y` granule. This is variable based on `at` (e.g., 28-31 days per month). Only implement this directly (rather than `chronon_cardinality_fixed()`) when the relationship genuinely depends on `at`, and pair it with a direct `chronon_divmod()` method so that the relationship remains reachable during graph traversal.
- `chronon_divmod(x, from, to)` - Converts time point `x` from granules of `from` to granules of `to`, returning a list with `div` (the quotient) and `mod`. This enables conversions between granules that have variable cardinality (e.g., the date 2020-03-23 to the month 2020-03). All conversions should be based on chronons since epoch (1970-01-01), in the UTC time zone.

These methods work together to enable mixtime to perform calendar-aware arithmetic, understanding that months have variable lengths and handling timezone-aware conversions.

See Also

`new_calendar()` for creating calendars from time units

Examples

```
# Create a timezone-aware unit class

# Use these units to create a calendar
my_calendar <- new_calendar(
  day = S7::new_class("tu_my_day", parent = mt_unit),
  month = S7::new_class("tu_my_month", parent = mt_tz_unit),
  class = "my_calendar"
)

# Access unit constructors from the calendar
my_calendar$day(1L)
my_calendar$month(3L, tz = "America/New_York")
```

---

new_calendar *Create a new calendar*

Description

Define a new calendar as a collection of time units. Calendars are the foundation for representing dates and times in terms of human-readable components like years, months, days, hours, minutes, and seconds. Each calendar is defined by specifying the time units it contains, which determine how time values can be interpreted and manipulated.

Usage

```
new_calendar(..., inherit = NULL, class = character())
```

Arguments

`...`	Named time unit class definitions. Each argument should be a time unit class (typically created with `S7::new_class()`) that inherits from `mt_unit` or `mt_tz_unit`. The names define the calendar's fields and are used to access unit constructors (e.g., `calendar$year()`).
`inherit`	Optional calendar to inherit time units from. Units defined in `...` will override inherited units with the same name.
`class`	Character vector of additional classes for the calendar object.

Details

Time units are typically S7 class definitions that inherit from `mt_unit` for standard units, `mt_tz_unit` for timezone-aware units (civil time), or `mt_loc_unit` for location-aware units (astronomical time). The calendar object provides a namespace for accessing these unit constructors and defines the relationships between them for calendar arithmetic.

Value

A calendar object of class `c(class, "mt_calendar")`, consisting of a named list containing the specified time unit classes.

See Also

`linear_time()`, `cyclical_time()`

Examples

```
# Create a simple calendar with year and month units
# (inheriting from civil time units for day, hour, minute, second, ...)
cal_simple <- new_calendar(
  year = new_time_unit("tu_year", parent = mt_tz_unit),
  month = new_time_unit("tu_month", parent = mt_tz_unit),
  inherit = cal_time_civil,
  class = "cal_simple"
)

# Create time granules from the calendar
cal_simple$year(1L)
cal_simple$month(1L)
```

`new_cyclical_time_fn` *Cyclical time function factory*

Description

`new_cyclical_time_fn()` creates a cyclical time function for a specified chronon and cycle. The cycle is the larger time granule that defines the time period over which the chronon loops (e.g., a week). The chronon is the smaller time granule that iterates within each cycle (e.g., a day). Combined, these two granules form a cyclical time relationship (e.g., day of the week).

Usage

```
new_cyclical_time_fn(chronon, cycle, default_calendar = cal_gregorian)
```

Arguments

`chronon`	A time granule object representing the chronon (e.g., `day(1L)`)
`cycle`	A time granule object representing the cycle (e.g., `week(1L)`)
`default_calendar`	A default calendar used to find the time units for conversion if they don't exist in the calendar of the input data (e.g., `cal_isoweek`)

Value

A function used to create cyclical time points with a specific chronon and cycle.

Examples

```
day_of_week <- new_cyclical_time_fn(day(1L), week(1L), default_calendar = cal_isoweek)
day_of_week(Sys.Date())

month_of_year <- new_cyclical_time_fn(month(1L), year(1L))
month_of_year(Sys.Date())
```

new_duration_fn *Duration function factory*

Description

new_duration_fn() creates a duration function for a specified chronon. A chronon is the smallest indivisible time unit (e.g., days, months) that defines what the numeric magnitudes in the resulting duration vector represent.

Usage

```
new_duration_fn(chronon, default_calendar = cal_gregorian)
```

Arguments

chronon	A bare call for a time unit object representing the chronon (e.g., month(1L), day(1L)).
default_calendar	A default calendar used to resolve the time units if they don't exist in the calendar of the input data (e.g., cal_gregorian).

Value

A function used to create duration vectors with a specific chronon. The returned function accepts:

data A numeric vector of duration magnitudes.

calendar A calendar system used to evaluate chronon. Defaults to time_calendar(data).

... Additional arguments passed to the chronon (e.g., tz for timezones).

See Also

- duration() for creating duration vectors directly
- cal_gregorian, cal_isoweek for calendar systems

Examples

```
# Create a months duration function
months <- new_duration_fn(month(1L), default_calendar = cal_gregorian)
months(1:6)

# Create a days duration function
days <- new_duration_fn(day(1L), default_calendar = cal_gregorian)
days(1:7)
```

`new_linear_time_fn` *Linear time function factory*

Description

`new_linear_time_fn()` creates a linear time function for a specified chronon. A chronon is the smallest indivisible time granule (e.g., days, hours).

Usage

```
new_linear_time_fn(chronon, default_calendar = cal_gregorian)
```

Arguments

`chronon`	A bare call for a time granule object representing the chronon (e.g., `day(1)`)
`default_calendar`	A default calendar used to find the time units for conversion if they don't exist in the calendar of the input data (e.g., `cal_isoweek` for week chronons to work with gregorian calendar inputs).

Value

A function used to create linear time points with a specific chronon.

Examples

```
# Linear time with 1 month granules as the chronon
ym <- new_linear_time_fn(month(1L))
ym(Sys.Date())

# Linear time with 1 day granules as the chronon
yd <- new_linear_time_fn(day(1L))
yd(Sys.Date())

# Linear time with 1 week granules as the chronon, using the ISO week calendar
yw <- new_linear_time_fn(week(1L), default_calendar = cal_isoweek)
yw(Sys.Date())

# Linear time with 1 hour granules as the chronon
ymd_h <- new_linear_time_fn(hour(1L))
ymd_h(Sys.time())
```

new_mixtime *Constructor for mixtime vectors*

Description

Creates a `mixtime` vector, which can contain time points of different granularities (e.g. monthly and quarterly) in a single vector via `vecvec`.

Usage

```
new_mixtime(x = mt_linear())
```

Arguments

`x`	A mixtime time vector (e.g. created with `mt_linear()`) to wrap in a mixtime class.

Value

A `mixtime` object, which allows mixed-type time vectors to coexist in a single vector.

new_time *Constructor for mixtime time vectors*

Description

[Defunct]

`new_time()` was the low-level constructor for `mt_time` vectors. It is now defunct; call the concrete time class constructors directly instead: `mt_linear()` for linear time, `mt_duration()` for durations, and `mt_cyclical()` for cyclical time.

Usage

```
new_time(x = integer(), chronon = mt_unit(1L), cycle = NULL, class = NULL)
```

Arguments

`x`, `chronon`, `cycle`, `class`	Ignored.

`new_time_unit` *Create a new time unit class*

Description

Define a new S7 class representing a time unit for use in a mixtime calendar. Time units are the building blocks of calendars: each unit represents a specific temporal component (e.g., day, month, year) and carries a step size `n`. Units are combined via `new_calendar()` to form a complete calendar system.

Usage

```
new_time_unit(
  name,
  parent = mt_unit,
  package = topNamespaceName(parent.frame()),
  properties = list(),
  abstract = FALSE,
  constructor = NULL,
  validator = NULL
)
```

Arguments

`name`	A string naming the new S7 class (e.g., `"tu_my_year"`). By convention, time unit class names are prefixed with `tu_`.
`parent`	The parent S7 class. Should be one of `mt_unit`, `mt_tz_unit`, or `mt_loc_unit`, or a class that itself inherits from one of these. Defaults to `mt_unit`.
`package`	A string giving the package name that owns the class. Defaults to the name of the calling namespace, so typically does not need to be set explicitly.
`properties`	A named list of additional `S7::new_property()` definitions beyond those inherited from `parent`. Each entry becomes a slot on instances of the new class. Defaults to an empty list.
`abstract`	Logical. If `TRUE` the class cannot be instantiated directly; it serves only as a base for further subclassing.
`constructor`	A function to use as the class constructor, or `NULL` (default) to generate one automatically. The auto-generated constructor accepts `...` (forwarded to the parent constructor) plus one argument per entry in `properties`, with defaults taken from each property's `default` field.
`validator`	A function of one argument (`self`) that returns `NULL` when the object is valid, or a character string describing the problem. See `S7::new_class()` for details.

Details

Choose the parent class based on the type of time the unit represents:

- `mt_unit` — abstract or calendar-only time (no timezone or location context; e.g., Gregorian year or ISO week).
- `mt_tz_unit` — civil time with a timezone (e.g., a clock hour that needs `tz` to resolve wall-clock ambiguity).
- `mt_loc_unit` — astronomical time with a geographic location (e.g., a solar day tied to observer longitude/latitude).

Value

An S7 class object, as returned by `S7::new_class()`.

See Also

- `mt_unit`, `mt_tz_unit`, `mt_loc_unit` for the base classes to inherit from.
- `new_calendar()` for assembling time units into a calendar.
- `S7::new_class()` for the underlying S7 class constructor.

Examples

```
# An abstract unit with no properties
tu_my_seq <- new_time_unit("tu_my_seq", parent = mt_unit)

# A civil-time unit with an extra property
tu_my_quarter <- new_time_unit(
  "tu_my_quarter",
  parent = mt_tz_unit,
  properties = list(
    fiscal = S7::new_property(S7::class_logical, default = FALSE)
  )
)
```

`parse_format` *Candidate format strings for parsing time*

Description

Combines one or more format strings into a character vector for use as the `format` argument of `time_parse()`. When multiple format strings are given, `time_parse()` tries each in turn and keeps whichever parses the most values, so `parse_format()` is the usual way to build up a set of candidates to try (it's also how `chronon_parse_linear()`/ `chronon_parse_cyclical()` methods build theirs). It can optionally mark its candidates as using regex syntax rather than literal text; see the `regex` argument below.

Usage

```
parse_format(..., regex = FALSE)
```

Arguments

`...`	Format strings, as for the `format` argument of `time_parse()`.
`regex`	Whether the literal (non-token) text in `...` should be matched as regular expression syntax rather than escaped literally (e.g. `"[/-]"` to accept either `/` or `-` as a separator). This is a niche option for irregular text - most formats leave it at the default, `FALSE`. See the `regex` argument of `time_parse()` for details.

Value

A character vector of format strings suitable for `time_parse()`, with a `"regex"` attribute for the `regex` argument the parser.

See Also

`time_parse()` for using the result as `format`, `chronon_parse_linear()`/`chronon_parse_cyclical()` for calendar-specific candidates built this way.

Examples

```
parse_format("{lin(year)}-{cyc(month, year)}-{cyc(day, month)}")

# Multiple candidates are tried in turn, keeping whichever parses most values
parse_format(
  "{lin(year)}-{cyc(month, year)}-{cyc(day, month)}",
  "{lin(year)}/{cyc(month, year)}/{cyc(day, month)}"
)

# regex = TRUE treats the surrounding text as regular expression syntax
parse_format(
  "{lin(year)}[/-]{cyc(month, year)}[/-]{cyc(day, month)}",
  regex = TRUE
)
```

---

seq.mixtime::mixtime *Generate sequences of mixtime values*

Description

Create regular sequences of time values. This method handles both linear time sequences (dates, date-times) and cyclical time sequences (day of week, month of year).

Usage

```
## S3 method for class '`mixtime::mixtime`'
seq(...)

## S3 method for class '`mixtime::mt_time`'
seq(
  from,
  to,
  by,
  length.out = NULL,
  along.with = NULL,
  on_invalid = getOption("mixtime.on_invalid", "clamp"),
  ...
)
```

Arguments

`...`	Additional arguments passed to the underlying sequence method.
`from`	Starting value of the sequence.
`to`	End value of the sequence (if provided).
`by`	Increment of the sequence. Can be: • A numeric for the number of time chronons • A character string specifying the interval (e.g., "1 day", "2 weeks", "1 month", "1 year") • A time granule object created with time unit functions (e.g., `cal_gregorian$year(1L)`, `cal_gregorian$month(1L)`, `cal_gregorian$day(1L)`) • A time `duration()` object (e.g., `years(1L)`, `months(1L)`, `days(1L)`)
`length.out`	Desired length of the sequence (alternative to `to`).
`along.with`	Take the length from this argument (alternative to `length.out`).
`on_invalid`	How to handle time points that don't exist, either because they overflow the cycle when using a by argument with different time granule than the sequence chronon, or because they fall in the local time skipped when daylight saving starts. Defaults to the `mixtime.on_invalid` option, or `"clamp"` if it isn't set. Options are: • `"clamp"`: Reduce the overflowing component to its last valid value, preserving finer components (e.g., the time of day). A skipped local time moves back by the size of the gap (02:30 becomes 01:30) • `"previous"`: Use the end of the cycle, so finer components are not preserved. For discrete time this is the last chronon within the cycle; for continuous time it is the end of that chronon, the same instant as `"next"`. A skipped local time becomes the end of the valid time before the gap (01:59:59) • `"next"`: Use the start of the following cycle, so finer components are not preserved. A skipped local time becomes the start of the valid time after the gap (03:00)

- `"overflow"`: Allow time points to overflow into the next cycle. A skipped local time moves forward by the size of the gap (02:30 becomes 03:30)
- `"reject"`: Return `NA` with a warning
- `"error"`: Throw an error

This is relevant when the starting time point has an offset that doesn't exist in all cycles. For example, starting on day 31 with `by = "1 month"` will overflow in months with fewer than 31 days (e.g., February). With `"clamp"`, these will be adjusted to the last day of the month (e.g., Feb 28/29). With `"next"`, they move to the first day of the next month, and with `"overflow"`, the extra days carry into the next month.

If neither `on_invalid` nor the `mixtime.on_invalid` option is set and overflow occurs, a warning is issued with the default `"clamp"` behavior applied. See `time_shift()` for shifting time points with the same options.

Details

For linear time types (Date, POSIXct, yearmonth, etc.), sequences progress forward or backward in time. For cyclical time types (month_of_year, day_of_week, etc.), sequences wrap around cyclically.

Each time point is `from` (or `to`, if `from` is missing) shifted by a whole number of `by` steps, matching `from + duration(k, by)`. For time-zone-aware times, steps of a day or longer keep the local clock time, while shorter steps (e.g. hours) are real time across daylight saving changes.

Value

A mixtime vector containing the sequence.

Examples

```
# Linear time sequences with integer by
seq(yearmonth("2020 Jan"), yearmonth("2020 Dec"))
seq(yearquarter("2020 Q1"), length.out = 5, by = 3)

# Linear time sequences with string intervals
seq(date("2020-01-01"), date("2020-12-31"), by = "1 month")
seq(yearmonth("2020 Jan"), yearmonth("2025 Jan"), by = "1 year")
seq(date("2020-01-01"), length.out = 10, by = "2 weeks")

# Linear time sequences incrementing by time granules
seq(yearmonth("2020 Jan"), yearmonth(("2020 Dec")), by = cal_gregorian$month(2L))
seq(date("2020-01-01"), length.out = 5, by = cal_gregorian$year(1L))
seq(date("2020-01-01"), date("2020-01-31"), by = cal_gregorian$day(7L))

# Handling invalid dates with on_invalid
seq(date("2020-01-31"), length.out = 3, by = "1 month")  # warns, uses clamp
seq(date("2020-01-31"), length.out = 3, by = "1 month", on_invalid = "clamp")
seq(date("2020-01-31"), length.out = 3, by = "1 month", on_invalid = "next")
seq(date("2020-01-31"), length.out = 3, by = "1 month", on_invalid = "overflow")

# Cyclical time sequences
```

```
seq(month_of_year(0L), month_of_year(11L))
seq(day_of_week(0L), day_of_week(6L), by = 1)
```

`time_calendar` *Obtain the calendar of a time object*

Description

This S7 generic function extracts the calendar system from a time object. The calendar defines the collection of time units (years, months, days, etc.) used to interpret the time representation.

Usage

```
time_calendar(x, ...)
```

Arguments

`x`	A time object (e.g., base::Date, base::POSIXct, `linear_time()`, etc.)
`...`	Additional arguments for methods.

Value

A calendar object (e.g., `cal_gregorian`, `cal_isoweek`)

Examples

```
# The calendar of a Date object is the Gregorian calendar
time_calendar(Sys.Date())

# The calendar of a POSIXct object is also Gregorian
time_calendar(Sys.time())

# The calendar of a yearweek object is the ISO week calendar
time_calendar(yearweek(Sys.Date()))

# A mixed time object returns a list of calendars
time_calendar(c(yearmonth(Sys.Date()), Sys.Date()))
```

`time_chronon` *Obtain the chronon of a time object*

Description

This S7 generic function extracts the chronon (the smallest time granule) from a time object, such as continuous time or cyclical time representations.

Usage

```
time_chronon(x, ...)
```

Arguments

`x`	A time object (e.g., base::Date, base::POSIXct, `linear_time()`, etc.)
`...`	Additional arguments for methods.

Details

For base::Date and base::POSIXct objects the chronon is identified from the greatest common divisor of the intervals between observations, in the same way as a tsibble's interval. For example, hourly datetimes have a 1 hour chronon rather than 1 second. A multiple of a unit (e.g. 3 hours) is only used if every observation falls on a boundary of that chronon; otherwise a single unit is used so that no information is lost when converting.

Value

A time `duration()` vector representing the chronon of each value (e.g., `days(1L)`).

Examples

```
# The chronon of a Date or POSIXct object is identified from the interval
# between observations (as with tsibble's interval)
time_chronon(Sys.Date() + 0:2)
time_chronon(as.POSIXct("2024-01-01", tz = "UTC") + 3600 * 0:2)

# Without an interval, the chronon of a Date is 1 day and a POSIXct is 1 second
time_chronon(Sys.Date())
time_chronon(Sys.time())

# The chronon of a continuous time year and month is 1 month
time_chronon(yearmonth(Sys.Date()))

# The common chronon of a mixed time object is the finest chronon
time_chronon(c(yearmonth(Sys.Date()), Sys.Date()))
```

`time_components` *Extract linear and cyclical time components*

Description

`time_components()` decomposes a time vector into its constituent parts using `dplyr::mutate()`-like semantics. Each named expression is built from the `lin()` and `cyc()` helpers (the same vocabulary used in `format()` strings) and produces a component time vector:

Usage

```
time_components(x, ..., calendar = time_calendar(x))
```

Arguments

`x`	A `mixtime` (or an object coercible to one via `as_mixtime()`, such as a `Date` or `POSIXct`).
`...`	Named expressions using `lin()` and `cyc()` describing the components to extract. The granule names (e.g. `year`, `month`, `day`) are resolved in the calendar of `x`.
`calendar`	Calendar system used to resolve granule names, overlaid on the calendar of `x`. Defaults to `time_calendar(x)`. Supply e.g. cal_isoweek to make ISO week-based components available.

Details

- `lin(<granule>)` extracts a **linear** component (a non-repeating count, e.g. the year), returning a linear time vector.
- `cyc(<granule>, <cycle>)` extracts a **cyclical** component (a repeating position within a larger cycle, e.g. the month within the year), returning a cyclical time vector.

All requested components are computed together in a single decomposition of the underlying time vector (via `chronon_parts()`), reusing the shared recursive `chronon_divmod()` results rather than converting each component independently.

Value

A data frame with one column per requested component. `lin()` columns are linear (`mt_linear`) time vectors and `cyc()` columns are cyclical (`mt_cyclical`) time vectors.

See Also

`lin()` and `cyc()` for the component helpers, `linear_time()` and `cyclical_time()` for constructing individual component vectors, and `format()` for the string counterpart of this interface.

Examples

```
t <- yearmonth(as.Date("2026-02-14") + c(0, 40, 400))

# Extract the year (linear) and month-of-year (cyclical)
time_components(t, yr = lin(year), mth = cyc(month, year))

# Components can be named automatically from the expression
time_components(as.Date("2025-12-15") + 0:3, cyc(day, cal_isoweek$week))
```

`time_compose` *Compose a linear time vector from linear and cyclical components*

Description

`time_compose()` is the inverse of `time_components()`: given a set of `lin()`/`cyc()` components it reconstructs the corresponding time points. Each component is either a two-sided formula pairing a spec with its value, or an already-tagged linear/cyclical time vector (e.g. produced by `linear_time()`, `cyclical_time()`, or a `time_components()` column).

Usage

```
time_compose(..., discrete = TRUE, calendar = cal_gregorian)
```

Arguments

`...`	Components used to build the time point. Each element is either: • a two-sided formula, `lin(<granule>) ~ <value>` or `cyc(<granule>, <cycle>) ~ <value>` (see `lin()`/`cyc()`), or • an existing linear or cyclical `mixtime` vector.
`discrete`	Logical. If `TRUE` (default), returns integer chronons since Unix epoch (discrete time model). If `FALSE`, returns fractional chronons allowing representation of fractional time granules (continuous time model).
`calendar`	Calendar used to resolve bare granule names in `lin()`/ `cyc()` formulas. Defaults to cal_gregorian.

Details

A `lin()` component (the **anchor**), when supplied, fixes the absolute position at some granule (e.g. the year). Every other component must be `cyc()`, chaining without gaps or branches from the anchor down to the target chronon: each cycle must equal another component's chronon exactly.

With no `lin()` anchor, every component must be `cyc()`, chained the same way but rooted at whichever component's cycle isn't itself another component's chronon. The result is cyclical time tagged with that root's cycle: `cyc(month, year) ~ 3` alone matches `month_of_year()` for any March; chaining `cyc(day, month) ~ 15` onto it collapses to one (day, year) pair, day-of-year 74, matching `day_of_year()` for 15 March.

Values of linear and cyclical components are specified on the right-hand-side of the formula. A `lin()` value is the real-world count (e.g. the literal year 1980); a `cyc()` value is the 1-indexed position within the cycle (e.g. `cyc(month, year) ~ 3` is the 3rd month, March), matching everyday counting rather than the raw 0-indexed position `time_components()` uses internally.

Value

A `mixtime` time vector, at the finest chronon reached by the chain (or the root's own chronon, if only one component is given). Linear with a `lin()` anchor, cyclical otherwise.

See Also

`time_components()` for the inverse operation, `lin()`/`cyc()` for the component vocabulary shared with `time_components()` and `format()`.

Examples

```
# cyc() values are 1-indexed positions: month 3 is March, day 15 is the 15th
time_compose(lin(year) ~ 1980, cyc(month, year) ~ 3, cyc(day, month) ~ 15)

# A lin() anchor alone is a valid (coarser) time point
time_compose(lin(year) ~ 1980)

# No lin() anchor: composes cyclical time
time_compose(cyc(month, year) ~ 3)

# Chaining collapses to one (chronon, cycle) pair: day 15 of month 3
# becomes day-of-year 74
time_compose(cyc(day, month) ~ 15, cyc(month, year) ~ 3)

# Round-tripping through time_components()
parts <- time_components(as.Date("2024-03-15"), yr = lin(year), mth = cyc(month, year))
with(parts, time_compose(yr, mth))

# Multi-unit (self-referencing) cycles: the 3rd month (1-indexed) of the
# 4th 3-month block since epoch (block 3 = months 9-11 -> December 1970)
time_compose(lin(month(3L)) ~ 3, cyc(month(1L), month(3L)) ~ 3)
```

---

time_cycle *Obtain the cycle of a time object*

Description

This S7 generic function extracts the cycle (the cyclical time granule) from a time object, such as cyclical time representations.

Usage

```
time_cycle(x, ...)
```

Arguments

x	A time object (e.g., base::Date, base::POSIXct, linear_time(), etc.)
...	Additional arguments for methods.

Value

A time duration() object representing the cycle of each value (e.g. weeks(1L)), or NA if the object has no cyclical component.

Examples

```
# Non-cyclical objects return NA
time_cycle(Sys.Date())

# The cycle of a cyclical time object
time_cycle(month_of_year(Sys.Date()))
```

---

time_is_complete_at *Test whether a granule is completed at a time point*

Description

time_is_complete_at() tests, for each element of a mixtime vector, whether the coarser granule that element falls into is fully observed *by the vector as a whole* – that is, whether every finer chronon making up that granule is present somewhere in x.

Usage

```
time_is_complete_at(x, granule, ...)
```

Arguments

x	A time object (typically a mixtime vector).
granule	The time granule whose precision to test, given as a granule generator (e.g. cal_gregorian$month) or a sized time unit (e.g. cal_gregorian$month(1L)).
...	Additional arguments for methods.

Details

Unlike time_is_determinate_at(), completeness is a collective property: an element is TRUE only when the other elements needed to fill its granule are also present. For example, in year(1L) the months of 2020 Jan : 2020 Oct are all FALSE (November and December are missing, so 2020 is incomplete), whereas in 2020 Jan : 2021 Mar the twelve months of 2020 are TRUE (they complete 2020) while the three months of 2021 remain FALSE.

A granule equal to x's own chronon is completed by each point on its own (TRUE). A granule finer than x cannot be completed by coarser points (FALSE). Missing (NA) and infinite times give NA.

Completeness is only defined within a single time granularity. Mixed-type `mixtime` vectors (e.g. months alongside days) are not yet supported and raise an error.

Value

A logical vector the same length as `x`.

See Also

`time_is_determinate_at()`

Examples

```
# 2020 Jan : 2020 Oct does not complete the year -> all FALSE
time_is_complete_at(yearmonth(as.Date("2020-01-01")) + 0:9, cal_gregorian$year(1L))

# 2020 Jan : 2021 Mar completes 2020 (TRUE) but not 2021 (FALSE)
time_is_complete_at(yearmonth(as.Date("2020-01-01")) + 0:14, cal_gregorian$year(1L))
```

---

time_is_determinate_at

Test whether time is determinate at a granule's precision

Description

`time_is_determinate_at()` tests, for each element of a `mixtime` vector, whether the time point is well-defined at the precision of `granule`.

Usage

```
time_is_determinate_at(x, granule, ...)
```

Arguments

`x`	A time object (typically a `mixtime` vector).
`granule`	The time granule whose precision to test, given as a granule generator (e.g. `cal_gregorian$month`) or a sized time unit (e.g. `cal_gregorian$month(1L)`).
`...`	Additional arguments for methods.

Details

Discrete (integer) time cannot resolve a granule finer than its own chronon (a `year()` has no determinate month), so those elements are `FALSE`. Continuous (fractional) time tracks progress within its chronon and so resolves finer granules exactly (0% through 2020 is 0% through January), giving `TRUE`. Coarser-or-equal granules are always determinate. Missing (`NA`) and infinite times give `NA`.

Value

A logical vector the same length as x.

Examples

```
# Discrete: a year has no determinate month
time_is_determinate_at(year(2020L), cal_gregorian$month(1L))

# Continuous: 0% through 2020 is 0% through January
time_is_determinate_at(year(2020), cal_gregorian$month(1L))

# A coarser granule is always determinate
time_is_determinate_at(yearmonth(as.Date("2020-02-01")), cal_gregorian$year(1L))
```

---

time_parse *Parse text into a time point*

Description

time_parse() is the inverse of format(): given text and the same {lin(...)}/{cyc(...)} template format uses, it reconstructs the time points that would have produced that text, via time_compose().

Usage

```
time_parse(
  x,
  chronon = NULL,
  cycle = NULL,
  format = NULL,
  regex = FALSE,
  na = c("", "NA"),
  calendar = NULL,
  locale = NULL,
  discrete = TRUE
)
```

Arguments

x	A character vector to parse.
chronon	Target time granule for the result, and (with cycle) the source of format candidates when format is NULL. Its attributes (e.g. tz) fill in whatever format leaves unset, and the result is converted onto it if format reaches a different chronon.
cycle	Target cycle granule, pairing with chronon for a cyclical result. Requires chronon.

format — A glue-style format string of lin()/cyc() tokens, e.g. "{lin(year)}-{cyc(month, year)}-{cyc(day, month)}" (see vignette("time-format-strings")), or several to try: whichever parses the most values of x is used for the whole vector (ties keep the earliest-listed format), and its unparsed values become NA (with a warning). Aborts if no format matches the shape of even one value. time_parse(format(x, fmt), format = fmt) round-trips back to x. NULL (the default) derives candidates from chronon/cycle via chronon_parse_linear()/chronon_parse_cyclic requires chronon.

regex — If FALSE (the default), literal text surrounding tokens is matched exactly. If TRUE, it's instead used verbatim as a regular expression, e.g. "[/-]" to accept either / or - as a separator; (...) groups you write are treated as non-capturing, since capturing groups are reserved for the tokens. Ignored when format is derived from chronon, which carries its own regex mode.

na — Strings to treat as missing (NA), checked before matching format. Not counted in the parsing-failure warning, unlike a value that fails to match format.

calendar — Calendar used to resolve granule names in format, and to disambiguate chronon's chronon_parse_linear() candidates when format is NULL. NULL (the default) uses time_calendar(cycle) or time_calendar(chronon), whichever is supplied, else cal_gregorian.

locale — Default locale for named (label = TRUE) tokens that don't specify their own. NULL defers to each token's own scheme.

discrete — Whether the result is discrete (integer chronon counts) or continuous (fractional). See linear_time().

Details

A format with a {lin(...)} token parses to linear time; one of only {cyc(...)} tokens parses to cyclical time, e.g. time_parse("Feb", format = "{cyc(month, year, label = TRUE)}") recovers the same kind of value as month_of_year().

Granule-specific extraction and decoding labels for each token is done by linear_labels_parse()/cyclical_labels_pars

Value

A mixtime time vector, the same length as x. Linear if format includes a {lin(...)} token (or cycle is NULL), cyclical otherwise.

See Also

format() for the inverse direction, time_compose() for composing a time point from already-decoded components, label_scheme() for declaring how a granule's labels parse, chronon_parse_linear()/chronon_par for the candidate formats derived from chronon/cycle, and vignette("time-format-strings") for the format string syntax.

Examples

```
time_parse("2024-02-15", format = "{lin(year)}-{cyc(month, year)}-{cyc(day, month)}")
time_parse(
  "15 Feb 2024",
```

```
  format = "{cyc(day, month)} {cyc(month, year, label = TRUE)} {lin(year)}"
)

# One bad value becomes NA (with a warning) instead of aborting the batch
time_parse(
  c("2024-02-15", "not a date"),
  format = "{lin(year)}-{cyc(month, year)}-{cyc(day, month)}"
)

# No {lin(...)} token: parses to cyclical time
time_parse("Feb", format = "{cyc(month, year, label = TRUE)}")

# Several formats: whichever parses the most values is used for the whole
# vector; here none of the "Y-M-D" format's values match, so the "D/M/Y"
# format (which matches both) is used instead
time_parse(
  c("15/02/2024", "20/03/2024"),
  format = c(
    "{lin(year)}-{cyc(month, year)}-{cyc(day, month)}",
    "{cyc(day, month)}/{cyc(month, year)}/{lin(year)}"
  )
)

# regex = TRUE: match "/" or "-" as the separator,
# and tolerate a trailing comment after the date.
time_parse(
  c("2024-02-15", "2024/02/15 (approx)"),
  format = "{lin(year)}[/-]{cyc(month, year)}[/-]{cyc(day, month)}( .*)?",
  regex = TRUE
)

# Default format strings from the target chronon, and results with `tz`.
time_parse("2024-02-15 09:00:00", chronon = cal_gregorian$second(1L, tz = "America/Los_Angeles"))
```

`time_round` *Round, floor and ceiling transformations for time objects*

Description

A family of helpers to round date/time objects to a specified time granule such as second, minute, hour, or day. These functions preserve the input time class, as rounded by the attributes of the `granule`.

Usage

```
time_round(x, granule, ...)

time_ceiling(x, granule, ...)

time_floor(x, granule, ...)
```

Arguments

x	A date/time object to be rounded. Accepted types include Date, POSIXct, POSIXlt and other objects that inherit from POSIXt. The returned object will be of the same class as the input.
granule	A time granule (or object coercible to a time granule, e.g. "day").
...	Additional arguments passed to specific implementations.

Value

An object of the same class as x with its time components adjusted to the requested granule.

See Also

base::round, lubridate::round_date

Examples

```
# Round POSIXct to the nearest minute (preserving tz)
t <- as.POSIXct("2020-01-01 12:34:56", tz = "UTC")
time_round(t, granule = cal_gregorian$minute(1L))

# Floor to the nearest hour
time_floor(t, granule = cal_gregorian$hour(1L))

# Ceiling a Date (treated as midnight-of-day rounding)
d <- as.Date("2020-01-01")
time_ceiling(d, granule = cal_gregorian$month(1L))
```

---

time_shift *Shift time points by a duration*

Description

Shifts time points by a duration, like x + by, with control over how time points that don't exist in the target granule are resolved. Shifting by a duration with an irregular relationship to the time points' chronon (e.g. days within months) can produce time points that don't exist, such as shifting 31 January by one month. In a time zone, shifting by days or longer keeps the local clock time, which may have been skipped when daylight saving starts (e.g. 02:30 on the day clocks go forward from 02:00 to 03:00). The + and - operators use the default strategy, set by the mixtime.on_invalid option ("clamp" unless set).

Usage

```
time_shift(x, by, on_invalid = getOption("mixtime.on_invalid", "clamp"))
```

Arguments

x	A time point vector.
by	A time duration() (e.g., months(1L)), or a character string specifying one (e.g., "1 month").
on_invalid	How to resolve time points that don't exist in the target granule. Defaults to the mixtime.on_invalid option, or "clamp" if it isn't set:

- "clamp": Reduce the invalid component to its last valid value, preserving finer components (e.g., the time of day). A skipped local time moves back by the size of the gap (02:30 becomes 01:30).
- "previous": Use the last valid time point before the invalid one, the end of the target granule, so finer components are not preserved. For discrete time this is the last chronon within the granule (e.g., 23:59:59 on the last day); for continuous time it is the end of that chronon, the same instant as "next". A skipped local time becomes the end of the valid time before the gap (01:59:59).
- "next": Use the first valid time point after the invalid one, the start of the following granule, so finer components are not preserved (e.g., 00:00:00 on the first day of the following month). A skipped local time becomes the start of the valid time after the gap (03:00).
- "overflow": Allow the invalid component to overflow into the next granule. A skipped local time moves forward by the size of the gap (02:30 becomes 03:30).
- "reject": Return NA with a warning.
- "error": Throw an error.

"previous" and "next" preserve the order of the time points being shifted, although distinct time points may become equal. "clamp" and "overflow" do not: shifting 1955-01-29 23:00:00 and 1955-01-31 08:00:00 by one month clamps them to 1955-02-28 23:00:00 and 1955-02-28 08:00:00, reversing their order.

Value

A time point vector of the same type as x.

Examples

```
x <- datetime("2021-01-30 09:30:00")
time_shift(x, months(1L))
time_shift(x, months(1L), on_invalid = "previous")
time_shift(x, months(1L), on_invalid = "next")
time_shift(x, months(1L), on_invalid = "overflow")
time_shift(x, months(1L), on_invalid = "reject")
try(time_shift(x, months(1L), on_invalid = "error"))

# Shifting backwards
time_shift(date("2021-03-31"), months(-1L), on_invalid = "overflow")
```

```
# 02:30 doesn't exist on 2025-10-05 in Melbourne (clocks go forward at 02:00)
x <- datetime(as.POSIXct("2025-10-04 02:30", tz = "Australia/Melbourne"))
time_shift(x, days(1L))
time_shift(x, days(1L), on_invalid = "previous")
time_shift(x, days(1L), on_invalid = "next")
time_shift(x, days(1L), on_invalid = "overflow")
```

`time_unit_full` *Time units as a string*

Description

These S7 generic functions provide the full and abbreviated names for time units. `time_unit_full()` is used in messages and durations (e.g., "2 months"). `time_unit_abbr()` is used for tsibble index interval displays (e.g., "1M"). `time_unit_plural()` pluralises the full unit name for a given quantity using cli-style pluralisation (e.g., `"year{?/s}"` becomes `"year"` or `"years"`).

Usage

```
time_unit_full(x, ...)

time_unit_plural(x, n = 2L)

time_unit_abbr(x, ...)
```

Arguments

`x`	A time granule object (e.g., `cal_gregorian$month(1L)`)
`...`	Additional arguments for methods.
`n`	Numeric quantity used to select singular or plural form.

Value

A string representing the time unit

Examples

```
time_unit_full(cal_gregorian$year(1L))
time_unit_plural(cal_gregorian$year(1L), 1L)
time_unit_plural(cal_gregorian$year(1L), 2L)
time_unit_abbr(cal_gregorian$year(1L))
```

tz_abbreviation *Get timezone abbreviation*

Description

Returns the timezone abbreviation (e.g., "EST", "PDT") for a given datetime in its specified timezone.

Usage

```
tz_abbreviation(x, tz = tz_name(x))
```

Arguments

x	A POSIXct datetime object or something coercible to POSIXct. The timezone is extracted from this object.
tz	A character vector of timezones to abbreviate at time point x.

Value

A character vector of timezone abbreviations.

Examples

```
tz_abbreviation(Sys.time())
tz_abbreviation(as.POSIXct("2024-01-15 12:00:00", tz = "America/New_York"))
```

tz_name *Extract timezone from an object*

Description

Generic function to extract the timezone from objects that have timezone information.

Usage

```
tz_name(x, ...)
```

Arguments

x	An object with timezone information.
...	Additional arguments passed to methods.

Value

A character vector representing the timezone of each time point (e.g., "America/New_York", "UTC").

Examples

```
tz_name(Sys.time())
tz_name(as.POSIXct("2024-06-15 12:00:00", tz = "America/New_York"))
```

`tz_offset` *Get timezone offset*

Description

Returns the UTC offset for a given datetime in its specified timezone.

Usage

```
tz_offset(x, ...)
```

Arguments

`x`	A time class coercible to POSIXt with an associated time zone.
`...`	Additional arguments passed to methods.

Value

A `mixtime` duration vector of offsets from UTC in the same chronon (e.g. seconds for POSIXt, days for dates, etc.)

Examples

```
tz_offset(as.POSIXct(Sys.time(), tz = Sys.timezone()))
tz_offset(as.POSIXct("2024-06-15 12:00:00", tz = "America/New_York"))
```

`tz_transitions` *Get timezone transitions*

Description

Returns all timezone transitions (e.g., daylight saving time changes) that occur between two datetimes. The timezone is taken from the start datetime.

Usage

```
tz_transitions(start, end)
```

Arguments

start	A POSIXct datetime object or something coercible to POSIXct, representing the start of the time range. The timezone is extracted from this object.
end	A POSIXct datetime object or something coercible to POSIXct, representing the end of the time range.

Value

A data frame with columns:

- time: A mixtime linear time point (continuous, UTC seconds) giving the instant of the transition.
- offset_before, offset_after: mixtime durations (UTC seconds) giving the UTC offset immediately before and after the transition.

Examples

```
# Get all DST transitions in 2024 for New York
tz_transitions(
  as.POSIXct("2024-01-01", tz = "America/New_York"),
  as.POSIXct("2024-12-31", tz = "America/New_York")
)
```

---

vocab_table *Build a* vocab *function from a plain name table*

Description

The common case for a label_scheme() vocab argument: a hand-listed table of names, one entry per locale, each a named list of renderings. By convention these follow CLDR's wide/abbreviated/narrow, but type can be anything. vocab_table() wraps it in the function(type = NULL, locale = NULL) shape vocab requires.

Usage

```
vocab_table(..., default_locale = "en-GB")
```

Arguments

...	Named entries, one per locale (e.g. `en-GB` = list(wide = month.name, abbreviated = month.abb)). Locale tags follow BCP 47.
default_locale	The locale returned when locale is unsupplied (or explicitly NULL, meaning "caller didn't ask", since time_parse()/format() always pass some value for locale).

Details

For vocab backed by an external i18n source instead of a hand-listed table, write that `function(type, locale)` directly instead. Same shape, no list needed.

Value

A function `function(type = NULL, locale = NULL)`, suitable as the vocab field of a `label_scheme()`.

See Also

`label_scheme()`

Examples

```
month_vocab <- vocab_table(`en-GB` = list(wide = month.name, abbreviated = month.abb))
month_vocab("abbreviated")
month_vocab(locale = "en-GB")
```

Index

!=,mt_cyclical-method
(mt_cyclical-compare), 42
!=,mt_duration-method
(mt_duration-compare), 44
!=,mt_linear-method
(mt_linear-compare), 45
<,mt_cyclical-method
(mt_cyclical-compare), 42
<,mt_duration-method
(mt_duration-compare), 44
<,mt_linear-method (mt_linear-compare),
45
<=,mt_cyclical-method
(mt_cyclical-compare), 42
<=,mt_duration-method
(mt_duration-compare), 44
<=,mt_linear-method
(mt_linear-compare), 45
==,mt_cyclical-method
(mt_cyclical-compare), 42
==,mt_duration-method
(mt_duration-compare), 44
==,mt_linear-method
(mt_linear-compare), 45
>,mt_cyclical-method
(mt_cyclical-compare), 42
>,mt_duration-method
(mt_duration-compare), 44
>,mt_linear-method (mt_linear-compare),
45
>=,mt_cyclical-method
(mt_cyclical-compare), 42
>=,mt_duration-method
(mt_duration-compare), 44
>=,mt_linear-method
(mt_linear-compare), 45
%d% (allen-interval-algebra), 3
%di% (allen-interval-algebra), 3
%f% (allen-interval-algebra), 3
%fi% (allen-interval-algebra), 3
%m% (allen-interval-algebra), 3
%mi% (allen-interval-algebra), 3
%o% (allen-interval-algebra), 3
%oi% (allen-interval-algebra), 3
%p% (allen-interval-algebra), 3
%pi% (allen-interval-algebra), 3
%s% (allen-interval-algebra), 3
%si% (allen-interval-algebra), 3

allen-interval-algebra, 3, *45*
as_mixtime, 6
as_mixtime(), *30*, *61*

base::Date, *39*, *59, 60*, *64*
base::POSIXct, *59, 60*, *64*
base::POSIXt, *39*
base::round, *69*

cal_gregorian, *23, 24*, *27, 28*, *37*, *39*, *51*, *62*,
67
cal_gregorian (calendar_gregorian), 6
cal_isoweek, *10*, *23, 24*, *27, 28*, *37*, *39*, *51*, *61*
cal_isoweek (calendar_isoweek), 8
cal_sym454 (calendar_sym454), 9
cal_time_civil, *10, 11*, *13*
cal_time_civil (calendar_time_civil), 10
cal_time_lunar, *13*
cal_time_lunar (calendar_time_lunar), 11
cal_time_lunar_synodic
(calendar_time_lunar), 11
cal_time_solar, *11*
cal_time_solar (calendar_time_solar), 12
cal_time_solar_transit
(calendar_time_solar), 12
calendar_gregorian, 6, *26*
calendar_isoweek, 8, *26*
calendar_sym454, 9
calendar_time_civil, 10
calendar_time_lunar, 11

calendar_time_solar, 12
chronon_cardinality, 14
chronon_cardinality(), *33–35*
chronon_common *(*chronon_glb*)*, 18
chronon_divmod, 15
chronon_divmod(), *48*
chronon_epoch, 16
chronon_epoch(), *36*
chronon_format_attr, 16
chronon_format_cyclical *(*chronon_format_linear*)*, 17
chronon_format_cyclical(), *19, 20*
chronon_format_duration *(*chronon_format_linear*)*, 17
chronon_format_linear, 17
chronon_format_linear(), *19, 20*
chronon_glb, 18
chronon_glb(), *18*
chronon_lub *(*chronon_glb*)*, 18
chronon_lub(), *18*
chronon_parse_cyclical *(*chronon_parse_linear*)*, 19
chronon_parse_cyclical(), *55, 56*, *67*
chronon_parse_linear, 19
chronon_parse_linear(), *55, 56*, *67*
circsum, 20
class_mixtime, 21
component_helpers, 22, *34*
cyc *(*component_helpers*)*, 22
cyc(), *61–63*, *67*
cyclical_labels *(*label_scheme*)*, 31
cyclical_labels(), *31, 32*, *35, 36*
cyclical_labels_format *(*linear_labels_format*)*, 35
cyclical_labels_format(), *33*, *35*
cyclical_labels_parse *(*linear_labels_format*)*, 35
cyclical_labels_parse(), *33*, *67*
cyclical_time, 23
cyclical_time(), *25, 26*, *46*, *49*, *61, 62*
cyclical_time_helpers, 25

date *(*linear_time_helpers*)*, 38
datetime *(*linear_time_helpers*)*, 38
day_of_month *(*cyclical_time_helpers*)*, 25
day_of_month(), *24*
day_of_week *(*cyclical_time_helpers*)*, 25
day_of_week(), *23, 24*
day_of_year *(*cyclical_time_helpers*)*, 25
day_of_year(), *24*
days *(*duration_helpers*)*, 27
duration, 26
duration(), *27, 28*, *46*, *51*, *57*, *60*, *64*, *70*
duration_helpers, 27

format(), *22*, *43*, *61*, *63*, *66, 67*
format,mt_time-method *(*format.mt_time*)*, 29
format.mt_time, 29

hours *(*duration_helpers*)*, 27

is_mixtime, 29
is_mixtime(), *6*
is_time, 30
is_time_cyclical *(*is_time*)*, 30
is_time_duration *(*is_time*)*, 30
is_time_linear *(*is_time*)*, 30

label_scheme, 31
label_scheme(), *31, 32*, *35, 36*, *67*, *74, 75*
lin *(*component_helpers*)*, 22
lin(), *61–63*, *67*
linear_labels *(*label_scheme*)*, 31
linear_labels(), *31, 32*, *35, 36*
linear_labels_format, 35
linear_labels_format(), *32*, *34, 35*
linear_labels_parse *(*linear_labels_format*)*, 35
linear_labels_parse(), *32*, *67*
linear_time, 37
linear_time(), *6–10*, *39*, *46*, *49*, *59–62*, *64*, *67*
linear_time_helpers, 38
loc_altitude *(*mixtime_location*)*, 42
loc_latitude *(*mixtime_location*)*, 42
loc_longitude *(*mixtime_location*)*, 42
lubridate::round_date, *69*

milliseconds *(*duration_helpers*)*, 27
minutes *(*duration_helpers*)*, 27
mixtime, *18*, 40
mixtime(), *5, 6*, *21, 22*, *30*, *43*, *45*
mixtime_location, 42
month_of_year *(*cyclical_time_helpers*)*, 25
month_of_year(), *23, 24*, *67*
months *(*duration_helpers*)*, 27

mt_cyclical *(mt_time-class)*, 46
mt_cyclical(), *46*, *53*
mt_cyclical-compare, *3–5*, 42
mt_duration *(mt_time-class)*, 46
mt_duration(), *46*, *53*
mt_duration-compare, 44
mt_linear, *43*
mt_linear *(mt_time-class)*, 46
mt_linear(), *21*, *46*, *53*
mt_linear-compare, *3–5*, *43*, 45
mt_loc_unit, *49*, *54, 55*
mt_loc_unit *(mt_unit)*, 47
mt_time *(mt_time-class)*, 46
mt_time-class, 46
mt_tz_unit, *49*, *54, 55*
mt_tz_unit *(mt_unit)*, 47
mt_unit, *46*, 47, *49*, *54, 55*

new_calendar, 48
new_calendar(), *47, 48*, *54, 55*
new_cyclical_time_fn, 50
new_cyclical_time_fn(), *24*, *26*
new_duration_fn, 51
new_duration_fn(), *27, 28*
new_linear_time_fn, 52
new_linear_time_fn(), *37*, *39*
new_mixtime, 53
new_mixtime(), *21, 22*
new_time, 53
new_time_unit, 54

parse_format, 55
parse_format(), *19, 20*

quarters *(duration_helpers)*, 27

S7::new_class(), *54, 55*
S7::new_property(), *54*
seconds *(duration_helpers)*, 27
seq.mixtime *(seq.mixtime::mixtime)*, 56
seq.mixtime::mixtime, 56
seq.mixtime::mt_time
(seq.mixtime::mixtime), 56

time_calendar, 59
time_calendar(), *17*
time_ceiling *(time_round)*, 68
time_chronon, 60
time_chronon(), *6*
time_components, 61
time_components(), *22*, *62, 63*
time_compose, 62
time_compose(), *66, 67*
time_cycle, 63
time_cycle(), *6*
time_floor *(time_round)*, 68
time_is_complete_at, 64
time_is_cyclical *(is_time)*, 30
time_is_determinate_at, 65
time_is_determinate_at(), *64, 65*
time_is_duration *(is_time)*, 30
time_is_linear *(is_time)*, 30
time_of_day *(cyclical_time_helpers)*, 25
time_parse, 66
time_parse(), *19, 20*, *55, 56*
time_round, 68
time_shift, 69
time_shift(), *58*
time_unit_abbr *(time_unit_full)*, 71
time_unit_full, 71
time_unit_plural *(time_unit_full)*, 71
trunc_time *(time_round)*, 68
tz_abbreviation, 72
tz_name, 72
tz_offset, 73
tz_transitions, 73

vctrs::vec_cast(), *6*
vec_cast(), *6*
vecvec::class_vecvec, *21*
vocab_table, 74
vocab_table(), *31–33*, *35, 36*

week_of_year *(cyclical_time_helpers)*, 25
week_of_year(), *24*
weeks *(duration_helpers)*, 27

year *(linear_time_helpers)*, 38
year(), *37*
yearmonth *(linear_time_helpers)*, 38
yearmonth(), *37*
yearquarter *(linear_time_helpers)*, 38
yearquarter(), *37*
years *(duration_helpers)*, 27
yearweek *(linear_time_helpers)*, 38
yearweek(), *8*, *37*