Package ‘almanac’

May 28, 2020

Title Tools for Working with Recurrence Rules

Version 0.1.1

Description Provides tools for defining recurrence rules and
recurrence bundles. Recurrence rules are a programmatic way to define
a recurring event, like the first Monday of December. Multiple
recurrence rules can be combined into larger recurrence bundles.
Together, these provide a system for adjusting and generating
sequences of dates while simultaneously skipping over dates in a
recurrence bundle's event set.

License MIT + file LICENSE

URL `https://github.com/DavisVaughan/almanac`

BugReports `https://github.com/DavisVaughan/almanac/issues`

Depends R (>= 3.2)

Imports glue, lubridate, magrittr, R6, rlang, V8 (>= 3.0.1), vctrs (>=
0.3.0)

Suggests covr, knitr, rmarkdown, testthat (>= 2.1.0)

VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 7.1.0

NeedsCompilation yes

Author Davis Vaughan [aut, cre],
RStudio [cph]

Maintainer Davis Vaughan <`davis@rstudio.com`>

Repository CRAN

Date/Publication 2020-05-28 17:20:24 UTC

R topics documented:

`adjustments` *Date adjustments*

Description

This family of adj_*() functions encode business logic for common date adjustments. If `x` falls on an event date, it is adjusted according to the function's adjustment rule. Otherwise it is left untouched.

- `adj_following()`

  Choose the first non-event date after `x`.

- `adj_preceding()`

  Choose the first non-event date before `x`.

- `adj_modified_following()`

  Choose the first non-event date after `x`, unless it falls in a different month, in which case the first non-event date before `x` is chosen instead.

- `adj_modified_preceding()`

  Choose the first non-event date before `x`, unless it falls in a different month, in which case the first non-event date after `x` is chosen instead.
- `adj_nearest()`

  Choose the nearest non-event date to `x`. If the closest preceding and following non-event dates are equally far away, the following non-event date is chosen.
- `adj_none()`

  Performs no adjustment and returns `x` unchanged.

Usage

```
adj_following(x, rschedule)

adj_preceding(x, rschedule)

adj_modified_following(x, rschedule)

adj_modified_preceding(x, rschedule)

adj_nearest(x, rschedule)

adj_none(x, rschedule)
```

Arguments

`x`	[Date] A vector of dates.
`rschedule`	[rschedule] An rschedule, such as an rrule or rbundle.

Value

An adjusted vector of Dates.

Examples

```
# A Saturday
x <- as.Date("1970-01-03")

on_weekends <- weekly() %>% recur_on_weekends()

# Adjust forward to Monday
adj_following(x, on_weekends)

# Adjust backwards to Friday
adj_preceding(x, on_weekends)

# Adjust to nearest non-event date
adj_nearest(x, on_weekends)
adj_nearest(x + 1, on_weekends)

# Sundays, one of which is at the end of the month
sundays <- as.Date(c("2020-05-24", "2020-05-31"))

# Adjust forward, unless that takes us into a new month, in which case we
# adjust backwards.
adj_modified_following(sundays, on_weekends)

# Saturdays, one of which is at the beginning of the month
saturdays <- as.Date(c("2020-08-01", "2020-08-08"))

# Adjust backwards, unless that takes us into a new month, in which
# case we adjust forwards
adj_modified_preceding(saturdays, on_weekends)
```

alma_events *Get all events*

Description

`alma_events()` retrieves all of the events in the rschedule's event set.

Usage

```
alma_events(rschedule)
```

Arguments

`rschedule`	[rschedule] An rschedule, such as an rrule or rbundle.

Value

A Date vector of events.

Examples

```
rrule <- daily(since = "1970-01-01", until = "1970-01-05")

alma_events(rrule)

rrule_weekly <- weekly(since = "1970-01-01") %>%
  recur_for_count(5)

rb <- runion() %>%
  add_rschedule(rrule) %>%
  add_rschedule(rrule_weekly)

alma_events(rb)
```

`alma_in` *Check if dates are in an event set*

Description

`alma_in()` checks if `x` is in the event set of dates defined by the rschedule.

Usage

```
alma_in(x, rschedule)
```

Arguments

`x`	[Date] A vector of dates.
`rschedule`	[rschedule] An rschedule, such as an rrule or rbundle.

Value

A logical vector the same size as `x`.

Examples

```
rrule <- weekly() %>%
  recur_on_wday("Thursday")

# A Thursday and Friday
x <- as.Date("1970-01-01") + 0:1

alma_in(x, rrule)

# Every month, on the 2nd day of the month
rrule2 <- monthly() %>%
  recur_on_mday(2)

# Make a larger rbundle made of multiple rules
rb <- runion() %>%
 add_rschedule(rrule) %>%
 add_rschedule(rrule2)

alma_in(x, rb)
```

alma_next *Generate the next or previous event*

Description

- `alma_next()` generates the next event after `x`.
- `alma_previous()` generates the previous event before `x`.

Usage

```
alma_next(x, rschedule, inclusive = FALSE)

alma_previous(x, rschedule, inclusive = FALSE)
```

Arguments

`x`	[Date] A vector of dates.
`rschedule`	[rschedule] An rschedule, such as an rrule or rbundle.
`inclusive`	[logical(1)] If `x` is an event, should it be considered the next or previous event?

Value

A Date vector the same size as `x`.

Examples

```
on_12th <- monthly() %>% recur_on_mday(12)
on_monday <- weekly() %>% recur_on_wday("Monday")

# On the 12th of the month, or on Mondays
rb <- runion() %>%
  add_rschedule(on_12th) %>%
  add_rschedule(on_monday)

alma_next(c("2019-01-01", "2019-01-11"), rb)
alma_previous(c("2019-01-01", "2019-01-11"), rb)
```

alma_search *Search for events*

Description

`alma_search()` retrieves all events between `from` and `to`.

Usage

```
alma_search(from, to, rschedule, inclusive = TRUE)
```

Arguments

`from, to`	[Date(1)] Dates defining the range to look for events.
`rschedule`	[rschedule] An rschedule, such as an rrule or rbundle.
`inclusive`	[logical(1)] If `from` or `to` are events, should they be included?

Value

A Date vector of all events between `from` and `to`.

Examples

```
on_12th <- monthly() %>% recur_on_mday(12)
on_monday <- weekly() %>% recur_on_wday("Monday")

# On the 12th of the month, or on Mondays
rb <- runion() %>%
  add_rschedule(on_12th) %>%
  add_rschedule(on_monday)

alma_search("2019-01-01", "2019-01-31", rb)
```

alma_seq *Generate date sequences*

Description

`alma_seq()` generates a sequence of all dates between `from` and `to`, skipping any events defined by the `rschedule`.

Usage

```
alma_seq(from, to, rschedule, inclusive = TRUE)
```

Arguments

`from, to`	[Date(1)] Dates defining the range to look for events.
`rschedule`	[rschedule] An rschedule, such as an rrule or rbundle.
`inclusive`	[logical(1)] If `from` or `to` are events in the `rschedule`, should they be removed from the sequence?

Value

A vector of dates in the range of [from, to], with all events in the `rschedule` removed.

Examples

```
on_weekends <- weekly() %>% recur_on_weekends()

# Generate a sequence of all non-weekend dates in Jan-2000
alma_seq("2000-01-01", "2000-01-31", on_weekends)
```

`alma_step` *Step relative to an rschedule*

Description

`alma_step()` is useful for shifting dates by "n business days".

`alma_step()` steps over a sequence of dates 1 day at a time, for `n` days. After each step, an adjustment is applied to shift to the next non-event date.

- If `n` is positive, `adj_following()` is called.
- If `n` is negative, `adj_preceding()` is called.
- If `n` is zero, it was arbitrarily decided to call `adj_following()` to roll to the next available non-event date.

Usage

```
alma_step(x, n, rschedule)
```

Arguments

`x`	[Date] A vector of dates.
`n`	[integer] The number of days to step. Can be negative to step backwards.
`rschedule`	[rschedule] An rschedule, such as an rrule or rbundle.

Details

Imagine you are on a Friday and want to shift forward 2 days using an rrule that marks weekends as events. `alma_step()` works like this:

- Step forward 1 day to Saturday.
- Apply an adjustment of `adj_following()`, which rolls forward to Monday.
- Step forward 1 day to Tuesday.
- Apply an adjustment of `adj_following()`, but no adjustment is required.

This lends itself naturally to business logic. Two business days from Friday is Tuesday.

Value

A Date vector the same size as `x` shifted by `n` steps.

Examples

```
# Make a rrule for weekends
on_weekends <- weekly() %>%
  recur_on_weekends()

# "Step forward by 2 business days"
# 2019-09-13 is a Friday.
# Here we:
# - Step 1 day to Saturday
# - Adjust to Monday
# - Step 1 day to Tuesday
alma_step("2019-09-13", 2, on_weekends)

# If Monday, 2019-09-16, was a recurring holiday, we could create
# a custom runion and step over that too.
on_09_16 <- yearly() %>%
  recur_on_ymonth(9) %>%
  recur_on_mday(16)

rb <- runion() %>%
  add_rschedule(on_09_16) %>%
  add_rschedule(on_weekends)

alma_step("2019-09-13", 2, rb)
```

`new-rbundle-set` *Constructor for a set-based recurrence bundle*

Description

These constructors are developer focused tools that are not required for normal usage of almanac. They construct new rbundle subclasses directly from a list of existing rschedules.

- `new_runion()` creates an runion.
- `new_rintersect()` creates an rintersect.
- `new_rsetdiff()` creates a rsetdiff.

Usage

```
new_rintersect(
  rschedules = list(),
  rdates = new_date(),
  exdates = new_date(),
  ...,
  class = character()
)

new_rsetdiff(
  rschedules = list(),
  rdates = new_date(),
  exdates = new_date(),
  ...,
  class = character()
)

new_runion(
  rschedules = list(),
  rdates = new_date(),
  exdates = new_date(),
  ...,
  class = character()
)
```

Arguments

`rschedules`	[list] A list of rschedules.
`rdates`	[Date] A vector of dates to forcibly include in the event set.
`exdates`	[Date] A vector of dates to forcibly exclude from the event set.
`...`	[named dots] Additional named elements added to the rbundle object.
`class`	[character] An optional subclass.

Value

A new rbundle subclass.

Examples

```
new_runion()

x <- daily()
y <- weekly()

rschedules <- list(x, y)

new_runion(rschedules)
```

`new_rbundle` *Constructor for an rbundle*

Description

`new_rbundle()` is a developer focused tool that is not required for normal usage of almanac. It constructs a new rbundle directly from a list of existing rschedules.

`rbundle_restore()` is a generic function that rbundle subclasses can provide a method for. It dispatches off of `to`. Its sole purpose is to restore classes and fields of the subclass after calling any of the following functions:

- `add_rdates()`
- `add_exdates()`
- `add_rschedule()`

Usage

```
new_rbundle(
  rschedules = list(),
  rdates = new_date(),
  exdates = new_date(),
  ...,
  class = character()
)

rbundle_restore(x, to)
```

Arguments

`rschedules`	[list] A list of rschedules.
`rdates`	[Date] A vector of dates to forcibly include in the event set.

`exdates`	[Date] A vector of dates to forcibly exclude from the event set.
`...`	[named dots] Additional named elements added to the rbundle object.
`class`	[character] An optional subclass.
`x`	[rbundle] An updated rbundle that needs to be restored to the type of `to`.
`to`	[rbundle subclass] An rbundle subclass that you are restoring to.

Details

An rbundle is an abstract class that rintersect, runion, and rsetdiff all inherit from. The sole purpose of an rbundle subclass is to implement an `rbundle_restore()` method that defines how to recover the original rbundle subclass after adding a new rschedule, rdate, or exdate. Additionally, because rbundles are also rschedules, a `rschedule_events()` method must be implemented.

Value

- `new_rbundle()` returns a new rbundle.
- `rbundle_restore()` should return an rbundle subclass of the same type as `to`.

Examples

```
new_rbundle()

x <- daily()
y <- weekly()

rschedules <- list(x, y)

new_rbundle(rschedules)
```

`new_rschedule` *Create a new rschedule*

Description

`new_rschedule()` is a developer focused tool that is not required for normal usage of almanac. It is only exported to allow other packages to construct new rschedule objects that work with almanac functions prefixed with alma_*(), like `alma_in()`.

`rschedule_events()` is a generic function that rschedule subclasses must provide a method for. `rschedule_events()` should return a Date vector containing the complete ordered set of events in the event set of that rschedule.

Usage

```
new_rschedule(..., class)

rschedule_events(x)
```

Arguments

`...`	[named fields] Named data fields.
`class`	[character] A required subclass.
`x`	[rschedule subclass] An object that subclasses rschedule.

Details

An rschedule is an abstract class that rrule and rbundle both inherit from. The sole functionality of rschedule classes is to provide a method for `rschedule_events()`.

Value

For `new_rschedule()`, a new rschedule subclass.

For `rschedule_events()`, a Date vector of events.

Examples

```
events <- as.Date("1970-01-01")

static <- new_rschedule(
  events = events,
  class = "static_rschedule"
)

# You have to register an `rschedule_events()` method first!
try(alma_events(static))
```

`radjusted` *Create an adjusted rschedule*

Description

`radjusted()` creates a new adjusted rschedule on top of an existing one. The new rschedule contains the same event dates as the existing rschedule, except when they intersect with the dates in the event set of the rschedule, `adjust_on`. In those cases, an `adjustment` is applied to the problematic dates to shift them to valid event dates.

This is most useful when creating corporate holiday rschedules. For example, Christmas always falls on December 25th, but if it falls on a Saturday, your company might observe Christmas on the previous Friday. If it falls on a Sunday, you might observe it on the following Monday. In this case, you could construct an rschedule for a recurring event of December 25th, and a second rschedule for weekends. When Christmas falls on a weekend, you would apply an adjustment of `adj_nearest()` to get the observance date.

Usage

```
radjusted(rschedule, adjust_on, adjustment)
```

Arguments

`rschedule`	[rschedule] An rschedule, such as an rrule or rbundle.
`adjust_on`	[rschedule] An rschedule that determines when the `adjustment` is to be applied.
`adjustment`	[function] An adjustment function to apply to problematic dates. Typically one of the pre-existing adjustment functions, like `adj_nearest()`. A custom adjustment function must have two arguments `x` and `rschedule`. `x` is the complete vector of dates that possibly need adjustment. `rschedule` is the rschedule who's event set determines when an adjustment needs to be applied. The function should adjust `x` as required and return the adjusted Date vector.

Value

An adjusted rschedule.

Examples

```
since <- "2000-01-01"
until <- "2010-01-01"

on_christmas <- yearly(since = since, until = until) %>%
  recur_on_ymonth("Dec") %>%
  recur_on_mday(25)

# All Christmas dates, with no adjustments
alma_events(on_christmas)

on_weekends <- weekly(since = since, until = until) %>%
  recur_on_weekends()

# Now all Christmas dates that fell on a weekend are
# adjusted either forwards or backwards, depending on which
# non-event date was closer
on_adj_christmas <- radjusted(on_christmas, on_weekends, adj_nearest)

alma_events(on_adj_christmas)
```

`rbundle-add` *Add to an rbundle*

Description

- `add_rschedule()` adds an rschedule to an rbundle. This can be another rrule or another rbundle.
- `add_rdates()` adds rdates to an rbundle. rdates are singular special cased dates that are forcibly included in the event set.
- `add_exdates()` adds exdates to an rbundle. exdates are singular special cased dates that are forcibly excluded from the event set.

Usage

```
add_rschedule(x, rschedule)

add_rdates(x, rdates)

add_exdates(x, exdates)
```

Arguments

`x`	[rbundle] An rbundle to add to.
`rschedule`	[rschedule] An rschedule, such as an rrule or rbundle.
`rdates`	[Date] Dates to forcibly include in the rbundle.
`exdates`	[Date] Dates to forcibly exclude from the rbundle.

Details

In terms of priority:

- An exdate will never be included.
- A rdate will always be included if it is not also an exdate.
- An event generated from an rschedule will always be included if it is not also an exdate.

Value

An updated rbundle.

Examples

```
on_thanksgiving <- yearly() %>%
  recur_on_wday("Thurs", 4) %>%
  recur_on_ymonth("Nov")

on_christmas <- yearly() %>%
  recur_on_mday(25) %>%
  recur_on_ymonth("Dec")

on_labor_day <- monthly() %>%
  recur_on_ymonth("Sep") %>%
  recur_on_wday("Mon", 1)

rb <- runion() %>%
  add_rschedule(on_thanksgiving) %>%
  add_rschedule(on_christmas) %>%
  add_rschedule(on_labor_day)

# Thanksgiving, Christmas, or Labor Day
alma_search("2019-01-01", "2021-01-01", rb)

# Except Labor Day in 2019
rb2 <- add_exdates(rb, "2019-09-02")

alma_search("2019-01-01", "2021-01-01", rb2)
```

`rbundle-set` *Create a new set-based recurrence bundle*

Description

Often, a single rrule will be sufficient. However, more complex recurrence objects can be constructed by combining multiple rschedules into a *recurrence bundle*.

There are three types of recurrence bundles provided in almanac, each of which construct their event sets by performing a set operation on the underlying event sets of the rschedules in the bundle.

- `runion()` takes the union.
- `rintersect()` takes the intersection.
- `rsetdiff()` takes the set difference.

Once you have created a recurrence bundle, you can:

- Add recurrence rules or other recurrence bundles with `add_rschedule()`.
- Forcibly include dates in its event set with `add_rdates()`.
- Forcibly exclude dates from its event set with `add_exdates()`.

Usage

```
rintersect()

rsetdiff()

runion()
```

Details

For `rsetdiff()`, the event set is created "from left to right" and depends on the order that the rschedules were added to the bundle.

Value

An empty rbundle.

See Also

`add_rschedule()`

Examples

```
since <- "2019-04-01"
until <- "2019-05-31"

on_weekends <- weekly(since = since, until = until) %>%
  recur_on_weekends()

on_25th <- monthly(since = since, until = until) %>%
  recur_on_mday(25)

# On weekends OR the 25th of the month
ru <- runion() %>%
  add_rschedule(on_weekends) %>%
  add_rschedule(on_25th)

alma_events(ru)

# On weekends AND the 25th of the month
ri <- rintersect() %>%
  add_rschedule(on_weekends) %>%
  add_rschedule(on_25th)

alma_events(ri)

# On weekends AND NOT the 25th of the month
rsd1 <- rsetdiff() %>%
  add_rschedule(on_weekends) %>%
  add_rschedule(on_25th)

alma_events(rsd1)
```

```
# On the 25th of the month AND NOT the weekend
rsd2 <- rsetdiff() %>%
  add_rschedule(on_25th) %>%
  add_rschedule(on_weekends)

alma_events(rsd2)
```

`recur_for_count` *Control the number of times to recur*

Description

`recur_for_count()` controls the total number of events in the recurrence set. Using `recur_for_count()` will override the `until` date of the rule.

Usage

```
recur_for_count(x, n)
```

Arguments

`x`	[rrule] A recurrence rule.
`n`	[positive integer(1)] The number of times to recur for.

Details

Remember that the number of times the occurrence has occurred is counted from the `since` date! Adjust it as necessary to get your desired results.

Value

An updated rrule.

Examples

```
# Using the default `since` date
daily_since_epoch_for_5 <- daily() %>% recur_for_count(5)

alma_search("1969-12-31", "1970-01-25", daily_since_epoch_for_5)

# Changing the `since` date
daily_since_2019_for_5 <- daily(since = "2019-01-01") %>% recur_for_count(5)

alma_search("2018-12-31", "2019-01-25", daily_since_2019_for_5)

# In the case of "impossible" dates, such as 2019-02-31 and 2019-04-31 in the
# example below, they are not added to the total count. Only true event
# dates are counted.
on_31_for_5 <- monthly(since = "2019-01-01") %>%
  recur_on_mday(31) %>%
  recur_for_count(5)

alma_search("2019-01-01", "2020-01-01", on_31_for_5)
```

`recur_on_easter` *Recur on easter*

Description

`recur_on_easter()` is a special helper to recur on Easter. Easter is particularly difficult to construct a recurrence rule for. Using `offset`, this can also be used to generate a recurrence rule on Easter Monday or Good Friday.

Usage

```
recur_on_easter(x, offset = 0L)
```

Arguments

`x`	[rrule] A recurrence rule.
`offset`	[integer(1)] An offset in terms of a number of days on either side of Easter to recur on. This offset must still fall within the same year, otherwise the date will be silently ignored.

Value

An updated rrule.

Examples

```
on_easter <- yearly() %>% recur_on_easter()
on_easter_monday <- yearly() %>% recur_on_easter(-1)

alma_search("1999-01-01", "2001-01-01", on_easter)

rb <- runion() %>%
  add_rschedule(on_easter) %>%
  add_rschedule(on_easter_monday)

alma_search("1999-01-01", "2001-01-01", rb)


# Note that `offset` must land within the same year, otherwise the date
# is ignored
on_easter_back_93_days <- yearly() %>% recur_on_easter(-93)
on_easter_back_94_days <- yearly() %>% recur_on_easter(-94)

alma_search("1999-01-01", "2001-01-01", on_easter_back_93_days)
alma_search("1999-01-01", "2001-01-01", on_easter_back_94_days)
```

`recur_on_interval` *Recur on an interval*

Description

`recur_on_interval()` adjusts the interval of the base frequency of the recurrence rule. For example, a `monthly()` rule with an interval of 2 would become "every other month".

Usage

```
recur_on_interval(x, n)
```

Arguments

`x`	[rrule] A recurrence rule.
`n`	[positive integer(1)] The interval on which to recur.

Value

An updated rrule.

Examples

```
# The default interval is 1
on_monthly <- monthly(since = "1999-01-01")

alma_search("1999-01-01", "1999-06-01", on_monthly)

# Adjust to every other month
on_every_other_month <- on_monthly %>% recur_on_interval(2)

alma_search("1999-01-01", "1999-06-01", on_every_other_month)

# Note that the frequency is limited to "every other month", but you
# can still have multiple events inside a single month
on_every_other_month_on_mday_25_or_26 <- on_every_other_month %>%
  recur_on_mday(25:26)

alma_search("1999-01-01", "1999-06-01", on_every_other_month_on_mday_25_or_26)
```

`recur_on_mday` *Recur on a day of the month*

Description

`recur_on_mday()` recurs on a specific day of the month.

Usage

```
recur_on_mday(x, mday)
```

Arguments

`x`	[rrule] A recurrence rule.
`mday`	[integer] The days of the month on which to recur. Negative values are allowed, which specify n days from the end of the month.

Details

If the day of the month doesn't exist for that particular month, then it is ignored. For example, if `recur_on_mday(30)` is set, then it will never generate an event in February.

Value

An updated rrule.

Examples

```
# When used with a yearly or monthly frequency, `recur_on_mday()` expands the
# number of days in the event set.
on_yearly <- yearly()
on_yearly_mday_1_to_2 <- on_yearly %>% recur_on_mday(1:2)

start <- "1999-01-01"
end <- "2000-06-30"

alma_search(start, end, on_yearly)
alma_search(start, end, on_yearly_mday_1_to_2)

# When used with a daily frequency, `recur_on_mday()` limits the number of
# days in the event set.
on_daily <- daily()
on_daily_mday_1_to_2 <- on_daily %>% recur_on_mday(1:2)

length(alma_search(start, end, on_daily))
length(alma_search(start, end, on_daily_mday_1_to_2))
```

```
# Using a negative value is a powerful way to look back from the end of the
# month. This is particularly useful because months don't have the same
# number of days.
on_last_of_month <- monthly() %>% recur_on_mday(-1)

alma_search(start, end, on_last_of_month)

# If you want particular days of the week at the end of the month, you
# could use something like this, which checks if the end of the month
# is also a Friday.
on_last_of_month_that_is_also_friday <- on_last_of_month %>% recur_on_wday("Friday")
alma_search(start, end, on_last_of_month_that_is_also_friday)

# But you probably wanted this, which takes the last friday of the month,
# on whatever day that lands on
on_last_friday_of_month <- monthly() %>% recur_on_wday("Friday", -1)
alma_search(start, end, on_last_friday_of_month)
```

recur_on_position *Recur on a position within a frequency*

Description

recur_on_position() let's you have fine tuned control over which element of the set to select *within* the base frequency.

Usage

```
recur_on_position(x, n)
```

Arguments

x	[rrule] A recurrence rule.
n	[integer] The positions to select within an intrafrequency set. Negative numbers select from the end of the set.

Value

An updated rrule.

Examples

```
library(lubridate, warn.conflicts = FALSE)

start <- "1999-01-01"
end <- "1999-05-01"
```

```
# You might want the last day of the month that is either a
# Sunday or a Monday, but you don't want to return both.
# This would return both:
on_last_monday_and_sunday <- monthly() %>%
  recur_on_wday(c("Monday", "Sunday"), -1)

alma_search(start, end, on_last_monday_and_sunday)

# To return just the last one, you would select the last value in
# the set, which is computed on a per month basis
on_very_last_monday_or_sunday <- on_last_monday_and_sunday %>%
  recur_on_position(-1)

alma_search(start, end, on_very_last_monday_or_sunday)

wday(alma_search(start, end, on_very_last_monday_or_sunday), label = TRUE)
```

`recur_on_wday` *Recur on a day of the week*

Description

- `recur_on_wday()` recurs on a specific day of the week.
- `recur_on_weekends()` and `recur_on_weekdays()` are helpers for recurring on weekends and weekdays.

Usage

```
recur_on_wday(x, wday, nth = NULL)

recur_on_weekdays(x)

recur_on_weekends(x)
```

Arguments

`x`	[rrule] A recurrence rule.
`wday`	[integer / character] Days of the week to recur on. Integer values must be from `1` to `7`, with `1` = `Monday` and `7` = `Sunday`. This is also allowed to be a full weekday string like `"Tuesday"`, or an abbreviation like `"Tues"`.
`nth`	[integer / NULL] Limit to the n-th occurrence of the `wday` in the base frequency. For example, in a monthly frequency, using `nth = -1` would limit to the last `wday` in the month. The default of `NULL` chooses all occurrences.

Details

Multiple week day values are allowed, and `nth` will be applied to all of them. If you want to apply different `nth` values to different days of the week, call `recur_on_wday()` twice with different `wday` values.

It is particularly important to pay attention to the `since` date when using weekly rules. The day of the week to use comes from the `since` date, which, by default, is a Monday (`1900-01-01`).

Value

An updated rrule.

Examples

```
# Using default `since` (1900-01-01, a Monday)
on_weekly_mondays <- weekly()

start <- "1999-01-01" # <- a Friday
end <- "1999-03-01"

# This finds the first Thursday, and then continues from there
alma_search(start, end, on_weekly_mondays)

# We start counting from a Friday here
on_weekly_fridays <- weekly(since = start)
alma_search(start, end, on_weekly_fridays)

# Alternatively, we could use `recur_on_wday()` and force a recurrence rule
# on Friday
on_wday_friday <- on_weekly_mondays %>% recur_on_wday("Friday")
alma_search(start, end, on_wday_friday)

# At monthly frequencies, you can use n-th values to look for particular
# week day events
on_first_friday_in_month <- monthly() %>% recur_on_wday("Fri", 1)
alma_search(start, end, on_first_friday_in_month)

# Negative values let you look from the back
on_last_friday_in_month <- monthly() %>% recur_on_wday("Fri", -1)
alma_search(start, end, on_last_friday_in_month)

# At yearly frequencies, this looks for the first sunday of the year
on_first_sunday_in_year <- yearly() %>% recur_on_wday("Sunday", 1)
alma_search(start, end, on_first_sunday_in_year)

# Last week day of the month
last_weekday_of_month <- monthly() %>%
  # Last occurrence of each weekday in the month
  recur_on_wday(c("Mon", "Tue", "Wed", "Thu", "Fri"), -1) %>%
  # Now choose the last one of those in each month
  recur_on_position(-1)
```

```
alma_search(start, end, last_weekday_of_month)
```

`recur_on_yday` *Recur on a day of the year*

Description

`recur_on_yday()` recurs on a specific day of the year.

Usage

```
recur_on_yday(x, yday)
```

Arguments

`x`	[rrule] A recurrence rule.
`yday`	[integer] Days of the year to recur on. Values must be from [-366, -1] and [1, 366].

Value

An updated rrule.

Examples

```
library(lubridate, warn.conflicts = FALSE)

on_5th_day_of_year <- yearly() %>% recur_on_yday(5)

alma_search("1999-01-01", "2000-12-31", on_5th_day_of_year)

# Notice that if you use a `since` date that has a day of the year
# after the specified one, it rolls to the next year
on_5th_day_of_year2 <- yearly(since = "1999-01-06") %>% recur_on_yday(5)
alma_search("1999-01-01", "2000-12-31", on_5th_day_of_year2)

# Negative values select from the back, which is useful in leap years
leap_year(as.Date("2000-01-01"))

last_day_of_year <- yearly() %>% recur_on_yday(-1)
last_day_of_year_bad <- yearly() %>% recur_on_yday(365)

alma_search("1999-01-01", "2000-12-31", last_day_of_year)
alma_search("1999-01-01", "2000-12-31", last_day_of_year_bad)
```

`recur_on_ymonth` *Recur on a month of the year*

Description

`recur_on_ymonth()` recurs on a specific month of the year.

Usage

```
recur_on_ymonth(x, ymonth)
```

Arguments

`x`	[rrule] A recurrence rule.
`ymonth`	[integer / character] Months of the year to mark as events. Integer values must be between [1, 12]. This can also be a full month string like `"November"`, or an abbreviation like `"Nov"`.

Value

An updated rrule.

Examples

```
# There is a big difference between adding this rule to a `yearly()`
# or `monthly()` frequency, and a `daily()` frequency.

# Limit from every day to every day in February
on_feb_daily <- daily() %>% recur_on_ymonth("Feb")

# Limit from 1 day per month to 1 day in February
on_feb_monthly <- monthly() %>% recur_on_ymonth("Feb")

start <- "1999-01-01"
end <- "2001-01-01"

alma_search(start, end, on_feb_daily)

alma_search(start, end, on_feb_monthly)
```

`recur_on_yweek` *Recur on a week of the year*

Description

`recur_on_yweek()` recurs on a specific week of the year.

Usage

```
recur_on_yweek(x, yweek)
```

Arguments

`x`	[rrule] A recurrence rule.
`yweek`	[integer] Weeks of the year to recur on. Integer values must be between [1, 53] or [-53, -1].

Details

Weekly rules are implemented according to the ISO-8601 standard. This requires that the first week of a year is the first one containing at least 4 days of the new year. Additionally, the week will start on the week day specified by `recur_with_week_start()`, which defaults to Monday.

Value

An updated rrule.

Examples

```
# Weekly rules are a bit tricky because they are implemented to comply
# with ISO-8601 standards, which require that the first week of the year
# is when there are at least 4 days in that year, and the week starts on
# the week day specified by `recur_with_week_start()` (Monday by default).
on_first_week <- yearly() %>% recur_on_yweek(1)

# In 2017:
# - Look at dates 1-4
# - 2017-01-02 is a Monday, so start the first week here
alma_search("2017-01-01", "2017-01-25", on_first_week)

# In 2015:
# - Look at dates 1-4
# - None of these are Monday, so the start of the week is
#   in the previous year
# - Look at 2014 and find the last Monday, 2014-12-29. This is the start of
#   the first week in 2015.
alma_search("2014-12-25", "2015-01-25", on_first_week)
```

```
# Say we want the start of the week to be Sunday instead of Monday!

# In 2015:
# - Look at dates 1-4
# - 2015-01-04 is a Sunday, so start the first week here
on_first_week_sun <- yearly() %>%
  recur_on_yweek(1) %>%
  recur_with_week_start("Sunday")

alma_search("2014-12-25", "2015-01-25", on_first_week_sun)
```

`recur_with_week_start` *Control the start of the week*

Description

`recur_with_week_start()` controls the week day that represents the start of the week. This is important for rules that use `recur_on_yweek()`.

The default day of the week to start on is Monday.

Usage

```
recur_with_week_start(x, wday)
```

Arguments

`x`	[rrule] A recurrence rule.
`wday`	[integer(1) / character(1)] Day of the week to start the week on. Must be an integer value in [1, 7], with `1 = Monday` and `7 = Sunday`. This is also allowed to be a full weekday string like `"Tuesday"`, or an abbreviation like `"Tues"`.

Value

An updated rrule.

Examples

```
# Weekly rules are a bit tricky because they are implemented to comply
# with ISO-8601 standards, which require that the first week of the year
# is when there are at least 4 days in that year, and the week starts on
# the week day specified by `recur_with_week_start()` (Monday by default).
on_first_week <- yearly() %>% recur_on_yweek(1)

# In 2017:
# - Look at dates 1-4
# - 2017-01-02 is a Monday, so start the first week here
```

```
alma_search("2017-01-01", "2017-01-25", on_first_week)

# In 2015:
# - Look at dates 1-4
# - None of these are Monday, so the start of the week is
#   in the previous year
# - Look at 2014 and find the last Monday, 2014-12-29. This is the start of
#   the first week in 2015.
alma_search("2014-12-25", "2015-01-25", on_first_week)

# Say we want the start of the week to be Sunday instead of Monday!

# In 2015:
# - Look at dates 1-4
# - 2015-01-04 is a Sunday, so start the first week here
on_first_week_sun <- yearly() %>%
  recur_on_yweek(1) %>%
  recur_with_week_start("Sunday")

alma_search("2014-12-25", "2015-01-25", on_first_week_sun)
```

`rrule` *Create a recurrence rule*

Description

These functions allow you to create a recurrence rule with a specified frequency. They are the base elements for all recurrence rules. To add to them, use one of the recur_*() functions.

- `daily()` Recur on a daily frequency.
- `weekly()` Recur on a weekly frequency.
- `monthly()` Recur on a monthly frequency.
- `yearly()` Recur on a yearly frequency.

Usage

```
daily(since = "1900-01-01", until = "2100-01-01")

weekly(since = "1900-01-01", until = "2100-01-01")

monthly(since = "1900-01-01", until = "2100-01-01")

yearly(since = "1900-01-01", until = "2100-01-01")
```

Arguments

`since`	[Date(1)] The lower bound on the event set. Depending on the final recurrence rule, pieces of information from this anchor date might be used to generate a complete recurrence rule.
`until`	[Date(1)] The upper bound on the event set.

Details

By default, `since == "1900-01-01"` and `until == "2100-01-01"`, which should capture most use cases well while still being performant. You may need to adjust these dates if you want events outside this range.

In terms of speed, it is generally more efficient if you adjust the `since` and `until` date to be closer to the first date in the sequence of dates that you are working with. For example, if you are working with dates in the range of 2019 and forward, adjust the `since` date to be `2019-01-01` for a significant speed boost.

As the anchor date, events are often calculated *relative to* this date. As an example, a rule of "on Monday, every other week" would use the `since` date to find the first Monday to start the recurrence from.

There is no `quarterly()` recurrence frequency, but this can be accomplished with `monthly() %>% recur_on_interval(3)`. The month to start the quarterly interval from will be pulled from the `since` date inside `monthly()`. The default will use a quarterly rule starting in January since the default `since` date is `1900-01-01`. See the examples.

Value

A new empty rrule.

Examples

```
rrule <- monthly() %>% recur_on_mday(25)

alma_search("1970-01-01", "1971-01-01", rrule)

# Notice that dates before 1900-01-01 are never generated with the defaults!
alma_search("1899-01-01", "1901-01-01", rrule)

# Adjust the `since` date to get access to these dates
rrule_pre_1900 <- monthly(since = "1850-01-01") %>% recur_on_mday(25)
alma_search("1899-01-01", "1901-01-01", rrule_pre_1900)

# A quarterly recurrence rule can be built from
# `monthly()` and `recur_on_interval()`
on_first_of_the_quarter <- monthly() %>%
  recur_on_interval(3) %>%
  recur_on_mday(1)

alma_search("1999-01-01", "2000-04-01", on_first_of_the_quarter)
```

```
# Alter the starting quarter by altering the `since` date
on_first_of_the_quarter_starting_in_feb <- monthly(since = "1998-02-01") %>%
  recur_on_interval(3) %>%
  recur_on_mday(1)

alma_search(
  "1999-01-01",
  "2000-04-01",
  on_first_of_the_quarter_starting_in_feb
)
```

`stepper` *Create a new stepper*

Description

- `stepper()` returns a function that can be used to add or subtract a number of days from a Date, "stepping" over events specified by an rschedule. You supply it the rschedule to step relative to, and then call the returned function with the number of days to step by.
- `workdays()` is a convenient stepper for stepping over the weekend.
- %s+% steps forwards.
- %s-% steps backwards.

You *must* use %s+ and %s-% to control the stepping. `+` and `-` will not work due to limitations in R's S3 dispatch system. Alternatively, you can call `vctrs::vec_arith()` directly, which powers %s+% with a correct double dispatch implementation.

Usage

```
stepper(rschedule)

x %s+% y

x %s-% y

workdays(n, since = "1900-01-01", until = "2100-01-01")
```

Arguments

`rschedule`	[rschedule] An rschedule, such as an rrule or rbundle.
`x, y`	[objects] Objects to perform step arithmetic on. Typically Dates or steppers.
`n`	[integer] The number of days to step. Can be negative to step backwards.

`since`	[Date(1)] The lower bound on the event set. Depending on the final recurrence rule, pieces of information from this anchor date might be used to generate a complete recurrence rule.
`until`	[Date(1)] The upper bound on the event set.

Details

Internally, a stepper is just powered by `alma_step()`, so feel free to use that directly.

Value

- `stepper()` returns a function of 1 argument, `n`, that can be used to step by `n` days, relative to the rschedule.
- `workdays()` return a new stepper object.
- %s+% and %s-% return a new shifted Date vector.

Examples

```
# A Thursday and Friday
x <- as.Date(c("1970-01-01", "1970-01-02"))

# Thursday is stepped forward 1 working day to Friday,
# and then 1 more working day to Monday.
# Friday is stepped forward 1 working day to Monday,
# and then 1 more working day to Tuesday
x %s+% workdays(2)

# ---------------------------------------------------------------------------

on_weekends <- weekly() %>%
  recur_on_weekends()

on_christmas <- yearly() %>%
  recur_on_mday(25) %>%
  recur_on_ymonth("Dec")

rb <- runion() %>%
  add_rschedule(on_weekends) %>%
  add_rschedule(on_christmas)

workday <- stepper(rb)

# Friday before Christmas, which was on a Monday
friday_before_christmas <- as.Date("2000-12-22")

# Steps over the weekend and Christmas to the following Tuesday
friday_before_christmas %s+% workday(1)

# ---------------------------------------------------------------------------
# Christmas in 2005 was on a Sunday, but your company probably "observed"
# it on Monday. So when you are on the Friday before Christmas in 2005,
# stepping forward 1 working day should go to Tuesday.

# We'll adjust the previous rule for Christmas to roll to the nearest
# non-weekend day, if it happened to fall on a weekend.
on_observed_christmas <- radjusted(
  on_christmas,
  adjust_on = on_weekends,
  adjustment = adj_nearest
)

# Note that the "observed" date for Christmas is the 26th
alma_search("2005-01-01", "2006-01-01", on_observed_christmas)

rb2 <- runion() %>%
  add_rschedule(on_weekends) %>%
  add_rschedule(on_observed_christmas)

workday2 <- stepper(rb2)

friday_before_christmas_2005 <- as.Date("2005-12-23")

# Steps over the weekend and the observed Christmas date
# of 2005-12-26 to Tuesday the 27th.
friday_before_christmas_2005 %s+% workday2(1)
```

Index