Package ‘MazamaTimeSeries’

September 22, 2023

Type Package
Version 0.2.15
Title Core Functionality for Environmental Time Series
Maintainer Jonathan Callahan <jonathan.s.callahan@gmail.com>
Description Utility functions for working with environmental time series data from known locations. The compact data model is structured as a list with two dataframes. A 'meta' dataframe contains spatial and measuring device metadata associated with deployments at known locations. A 'data' dataframe contains a 'datetime' column followed by columns of measurements associated with each ``device-deployment''. Ephemerides calculations are based on code originally found in NOAA's ``Solar Calculator''<https://gml.noaa.gov/grad/solcalc/>.
License GPL-3
URL https://github.com/MazamaScience/MazamaTimeSeries
BugReports https://github.com/MazamaScience/MazamaTimeSeries/issues
Depends R (>= 4.0.0)
Imports dplyr, geodist, lubridate, magrittr, methods, MazamaCoreUtils
 >= 0.4.15), MazamaRollUtils (>= 0.1.3), rlang, stringr
Suggests knitr, markdown, testthat (>= 2.1.0), rmarkdown, roxygen2
Encoding UTF-8
VignetteBuilder knitr
LazyData true
RoxygenNote 7.2.3
NeedsCompilation no
Author Jonathan Callahan [aut, cre], Hans Martin [ctb], Eli Grosman [ctb], Roger Bivand [ctb], Sebastian Luque [ctb]
Repository CRAN
Date/Publication 2023-09-22 07:20:12 UTC
### R topics documented:

- Camp_Fire .................................................. 3
- Carmel_Valley ............................................. 4
- example_mts ............................................... 4
- example_raws ............................................. 5
- example_sts ............................................... 6
- MazamaTimeSeries ....................................... 6
- mts_arrange ............................................... 7
- mts_check ................................................ 7
- mts_collapse ............................................. 8
- mts_combine .............................................. 10
- mts_distinct ............................................ 11
- mts_extractDataFrame .................................. 12
- mts_filterData .......................................... 12
- mts_filterDate .......................................... 13
- mts_filterDatetime ..................................... 15
- mts_filterMeta .......................................... 16
- mts_getDistance ........................................ 17
- mts_isEmpty ............................................ 18
- mts_isValid ............................................. 19
- mts_sample .............................................. 20
- mts_select .............................................. 21
- mts_selectWhere ....................................... 22
- mts_setTimeAxis ....................................... 23
- mts_slice_head ......................................... 24
- mts_summarize ......................................... 25
- mts_trim ................................................ 27
- mts_trimDate ........................................... 27
- requiredMetaNames ..................................... 28
- sts_check ............................................... 29
- sts_combine ............................................ 30
- sts_distinct ............................................ 31
- sts_extractDataFrame .................................. 32
- sts_filter .............................................. 32
- sts_filterDate ......................................... 33
- sts_filterDatetime ..................................... 35
- sts_isEmpty ............................................ 36
- sts_isValid ............................................. 37
- sts_summarize ......................................... 38
- sts_trimDate ........................................... 39
- timeInfo ............................................... 40

Index ......................................................... 43
**Camp_Fire**

**Camp Fire example dataset**

---

**Description**

The Camp_Fire dataset provides a quickly loadable version of a `mts_monitor` object for practicing and code examples.

**Usage**

```
Camp_Fire
```

**Format**

A `mts` object with 360 rows and 134 columns of data.

**Details**

The 2018 Camp Fire was the deadliest and most destructive wildfire in California’s history, and the most expensive natural disaster in the world in 2018 in terms of insured losses. The fire caused at least 85 civilian fatalities and injured 12 civilians and five firefighters. It covered an area of 153,336 acres and destroyed more than 18,000 structures, most with the first 4 hours. Smoke from the fire resulted in the worst air pollution ever for the San Francisco Bay Area and Sacramento Valley.

This dataset was generated on 2022-10-12 by running:

```r
library(AirMonitor)

Camp_Fire <-
  monitor_loadAnnual(2018) %>%
  monitor_filter(stateCode == 'CA') %>%
  monitor_filterDate(
    startdate = 20181108,
    enddate = 20181123,
    timezone = "America/Los_Angeles"
  ) %>%
  monitor_dropEmpty()

save(Camp_Fire, file = "data/Camp_Fire.rda")
```
**Carmel Valley**

*Carmel Valley example dataset*

**Description**

The Carmel Valley dataset provides a quickly loadable version of a single-sensor `mts_monitor` object for practicing and code examples.

**Usage**

Carmel_Valley

**Format**

An `mts` object with 600 rows and 2 columns of data.

**Details**

In August of 2016, the Soberanes fire in California burned along the Big Sur coast. It was at the time the most expensive wildfire in US history. This dataset contains PM2.5 monitoring data for the monitor in Carmel Valley which shows heavy smoke as well as strong diurnal cycles associated with sea breezes. Data are stored as an `mts` object and are used in some examples in the package documentation.

This dataset was generated on 2022-10-12 by running:

```r
library(AirMonitor)

Carmel_Valley <-
  airnow_loadAnnual(2016) %>%
  monitor_filterMeta(deviceDeploymentID == "a9572a904a4ed46d_840060530002") %>%
  monitor_filterDate(20160722, 20160815)

save(Carmel_Valley, file = "data/Carmel_Valley.rda")
```

---

**Example**

`mts` dataset

**Description**

The example_mts dataset provides a quickly loadable version of an `mts` object for practicing and code examples.

This dataset was generated on 2021-10-07 by running:

```r
```

---
library(AirSensor)

communities <- c("Alhambra/Monterey Park", "El Monte")

example_mts <-
  example_sensor_scaqmd %>%
  sensor_filterMeta(communityRegion %in% communities)

# Add required "locationName"
example_mts$meta$locationName <- example_mts$meta$siteName

save(example_mts, file = "data/example_mts.rda")

Usage

example_mts

Format

An mts object composed of "meta" and "data" dataframes.

Example RAWS dataset

The example_raws dataset provides a quickly loadable example of the data generated by the **RAWSmet** package. This data is a sts object containing hourly measurements from a RAWS weather station in Saddle Mountain, WA, between July 2002 and December 2017.

This dataset was generated on 2022-02-17 by running:

library(RAWSmet)

setRawsDataDir("~/Data/RAWS")

example_raws <-
  cefa_load(nwsID = "452701") %>%
  raws_filterDate(20160701, 20161001)

save(example_raws, file = "data/example_raws.rda")

Usage

example_raws

Format

An sts object composed of "meta" and "data" dataframes.
**Description**

The example_sts dataset provides a quickly loadable version of an sts object for practicing and code examples.

This dataset was was generated on 2021-01-08 by running:

```r
library(AirSensor)
example_sts <- example_pat
example_sts$meta$elevation <- as.numeric(NA)
example_sts$meta$locationName <- example_sts$meta$label

save(example_sts, file = "data/example_sts.rda")
```

**Usage**

`example_sts`

**Format**

An sts object composed of "meta" and "data" dataframes.
**mts_arrange**

Order mts time series by metadata values

**Description**

The variable(s) in ... are used to specify columns of mts$meta to use for ordering. Under the hood, this function uses arrange on mts$meta and then reorders mts$data to match.

**Usage**

mts_arrange(mts, ...)

**Arguments**

- **mts**  
  *mts* object.

- **...**  
  Variables in mts$meta.

**Value**

A reordered version of the incoming *mts* time series object. (A list with meta and data dataframes.)

**Examples**

```r
library(MazamaTimeSeries)

example_mts$meta$latitude[1:10]

# Filter for all labels with "SCSH"
byElevation <-
  example_mts %>%
  mts_arrange(latitude)

byElevation$meta$latitude[1:10]
```

**mts_check**

Check mts object for validity

**Description**

Checks on the validity of an *mts* object. If any test fails, this function will stop with a warning message.

**Usage**

mts_check(mts)
mts_collapse

Arguments

mts object.

Value

Returns TRUE invisibly if the mts object is valid.

See Also

mts_isValid

Examples

library(MazamaTimeSeries)
sts_check(example_mts)

# This would throw an error
if ( FALSE ) {

  broken_mts <- example_mts
  names(broken_mts) <- c('meta', 'bop')
  sts_check(broken_mts)
}

mts_collapse

Collapse an mts time series object into a single time series

Description

Collapses data from all time series in mts into a single-time series mts object using the function provided in the FUN argument. The single-time series result will be located at the mean longitude and latitude unless longitude and latitude are specified.

Any columns of mts$meta that are constant across all records will be retained in the returned mts$meta.

The core metadata associated with this location (e.g. countryCode, stateCode, timezone, ...) will be determined from the most common (or average) value found in mts$meta. This will be a reasonable assumption for the vast majority of intended use cases where data from multiple devices in close proximity are averaged together.
**Usage**

```r
mts_collapse(
    mts,
    longitude = NULL,
    latitude = NULL,
    deviceID = "generatedID",
    FUN = mean,
    na.rm = TRUE,
    ...
)
```

**Arguments**

- `mts`: `mts` object.
- `longitude`: Longitude of the collapsed time series.
- `latitude`: Latitude of the collapsed time series.
- `deviceID`: Device identifier for the collapsed time series.
- `FUN`: Function used to collapse multiple time series.
- `na.rm`: Logical specifying whether NA values should be ignored when `FUN` is applied.
- `...`: Additional arguments to be passed on to the `apply()` function.

**Value**

An `mts` time series object representing a single time series. (A list with `meta` and `data` dataframes.)

**Note**

After `FUN` is applied, values of `+/-Inf` and `NaN` are converted to `NA`. This is a convenience for the common case where `FUN = min/max` or `FUN = mean` and some of the time steps have all missing values. See the R documentation for `min` for an explanation.

**Examples**

```r
library(MazamaTimeSeries)

mon <-
    mts_collapse(
        mts = example_mts,
        deviceID = "example_ID"
    )

# mon$data now only has 2 columns
names(mon$data)

plot(mon$data, type = 'b', main = mon$meta$deviceID)
```
### mts_combine

**Combine multiple mts time series objects**

**Description**

Create a combined *mts* from any number of *mts* objects or from a list of *mts* objects. The resulting *mts* object will contain all deviceDeploymentIDs found in any incoming *mts* and will have a regular time axis covering the entire range of incoming data.

If incoming time ranges are non-contiguous, the resulting *mts* will have gaps filled with NA values.

An error is generated if the incoming *mts* objects have non-identical metadata for the same deviceDeploymentID unless replaceMeta = TRUE.

**Usage**

```r
mts_combine(
  ..., 
  replaceMeta = FALSE,
  overlapStrategy = c("replace all", "replace na")
)
```

**Arguments**

- ... Any number of valid *mts* objects.
- replaceMeta Logical specifying whether to allow replacement of metadata associated with deviceDeploymentIDs.
- overlapStrategy Strategy to use when data found in time series overlaps.

**Value**

An *mts* time series object containing all time series found in the incoming *mts* objects. (A list with meta and data dataframes.)

**Note**

Data for any deviceDeploymentIDs shared among *mts* objects are combined with a "later is better" sensibility where any data overlaps exist. To handle this, incoming *mts* objects are first split into "shared" and "unshared" parts.

Any "shared" parts are ordered based on the time stamp of their last record. Then dplyr::distinct() is used to remove records with duplicate datetime fields.

With overlapStrategy = "replace all", any data records found in "later" *mts* objects are preferentially retained before the "shared" data are finally reordered by ascending datetime.

With overlapStrategy = "replace missing", only missing values in "earlier" *mts* objects are replaced with data records from "later" time series.

The final step is combining the "shared" and "unshared" parts and placing them on a uniform time axis.
**Examples**

```r
library(MazamaTimeSeries)

ids1 <- example_mts$meta$deviceDeploymentID[1:5]
ids2 <- example_mts$meta$deviceDeploymentID[4:6]
ids3 <- example_mts$meta$deviceDeploymentID[8:10]

mts1 <-
  example_mts %>%
  mts_filterMeta(deviceDeploymentID %in% ids1) %>%
  mts_filterDate(20190701, 20190703)

mts2 <-
  example_mts %>%
  mts_filterMeta(deviceDeploymentID %in% ids2) %>%
  mts_filterDate(20190704, 20190706)

mts3 <-
  example_mts %>%
  mts_filterMeta(deviceDeploymentID %in% ids3) %>%
  mts_filterDate(20190705, 20190708)

mts <- mts_combine(mts1, mts2, mts3)

# Should have 1:6 + 8:10 = 9 meta records and the full date range
nrow(mts$meta)
range(mts$data$datetime)
```

---

**mts_distinct**

*Retain only distinct data records in mts$data*

**Description**

This function is primarily for internal use.

Two successive steps are used to guarantee that the datetime axis contains no repeated values:

1. remove any duplicate records
2. guarantee that rows are in datetime order

**Usage**

`mts_distinct(mts)`

**Arguments**

- `mts` *mts object*
**Value**

An `mts` object where each record is associated with a unique time. (A list with `meta` and `data` dataframes.)

---

**Description**

These functions are convenient wrappers for extracting the dataframes that comprise an `mts` object. These functions are designed to be useful when manipulating data in a pipeline chain using `%>%`.  

- `mts_extractData(mts)` is equivalent to `mts$data`.
- `mts_extractMeta(mts)` is equivalent to `mts$meta`.

**Usage**

```r
mts_extractData(mts)

mts_extractMeta(mts)
```

**Arguments**

- `mts` : `mts` object to extract dataframe from.

**Value**

A dataframe from the `mts` object.

---

**mts_filterData**

*General purpose data filtering for mts time series objects*

**Description**

A generalized data filter for `mts` objects to choose rows/cases where conditions are true. Multiple conditions may be combined with & or separated by a comma. Only rows where the condition evaluates to TRUE are kept. Rows where the condition evaluates to `NA` are dropped.

**Usage**

```r
mts_filterData(mts, ...)
```

**Arguments**

- `mts` : `mts` object.
- `...` : Logical predicates defined in terms of the variables in `mts$data`. 
Value

A subset of the incoming \textit{mts} time series object. (A list with meta and data dataframes.)

Note

Filtering is done on variables in \textit{mts$data} and results in an \textit{incomplete and irregular time axis}.

See Also

- \texttt{mts_filterDate}
- \texttt{mts_filterDatetime}
- \texttt{mts_filterMeta}

Examples

```r
library(MazamaTimeSeries)

# Are there any times when data exceeded 150?
sapply(example_mts$data, function(x) { any(x > 150, na.rm = TRUE) })

# Show all times where da4cadd2d6ea5302_4686 > 150
example_mts %>%
  mts_filterData(da4cadd2d6ea5302_4686 > 150) %>%
  mts_extractData() %>%
  dplyr::pull(datetime)
```

**mts_filterDate**

\textit{Date filtering for mts time series objects}

Description

Subsets an \textit{mts} object by date. This function always filters to day-boundaries. For sub-day filtering, use \texttt{mts_filterDatetime()}. Dates can be anything that is understood by \texttt{MazamaCoreUtils::parseDatetime()} including either of the following recommended formats:

- "YYYYmmdd"
- "YYYY-mm-dd"

Timezone determination precedence assumes that if you are passing in POSIXct values then you know what you are doing:

1. get timezone from \texttt{startdate} if it is POSIXct
2. use passed in timezone
3. get timezone from \texttt{mts}
mts_filterDate

Usage

mts_filterDate(
  mts = NULL,
  startdate = NULL,
  enddate = NULL,
  timezone = NULL,
  unit = "sec",
  ceilingStart = FALSE,
  ceilingEnd = FALSE
)

Arguments

mts
mts object.
startdate
Desired start date (ISO 8601).
enddate
Desired end date (ISO 8601).
timezone
Olson timezone used to interpret dates.
unit
Units used to determine time at end-of-day.
ceilingStart
Logical instruction to apply ceiling_date to the startdate rather than floor_date.
ceilingEnd
Logical instruction to apply ceiling_date to the enddate rather than floor_date.

Value

A subset of the incoming mts time series object. (A list with meta and data dataframes.)

Note

The returned data will run from the beginning of startdate until the beginning of enddate – i.e. no values associated with enddate will be returned. The exception being when enddate is less than 24 hours after startdate. In that case, a single day is returned.

See Also

mts_filterData
mts_filterDatetime
mts_filterMeta

Examples

library(MazamaTimeSeries)

example_mts %>%
  mts_filterDate(
    startdate = 20190703,
    enddate = 20190706
  ) %>%
  mts_extractData() %>%
mts_filterDatetime

```r
dplyr::pull(datetime) %>% range()
```

---

**mts_filterDatetime**  
*Datetime filtering for mts time series objects*

**Description**

DEPRECATED -- use `mts_setTimeAxis`.

Subsets an mts object by datetime. This function allows for sub-day filtering as opposed to `mts_filterDate()` which always filters to day-boundaries. Both the startdate and the enddate will be included in the subset.

Datetimes can be anything that is understood by `MazamaCoreUtils::parseDatetime()`. For non-POSIXct values, the recommended format is "YYYY-mm-dd HH:MM:SS".

Timezone determination precedence assumes that if you are passing in POSIXct values then you know what you are doing:

1. get timezone from `startdate` if it is POSIXct
2. use passed in timezone
3. get timezone from mts

**Usage**

```r
mts_filterDatetime(
  mts = NULL,
  startdate = NULL,
  enddate = NULL,
  timezone = NULL,
  unit = "sec",
  ceilingStart = FALSE,
  ceilingEnd = FALSE,
  includeEnd = FALSE
)
```

**Arguments**

- **mts**  
  *mts* object.

- **startdate**  
  Desired start datetime (ISO 8601).

- **enddate**  
  Desired end datetime (ISO 8601).

- **timezone**  
  Olson timezone used to interpret dates.

- **unit**  
  Datetimes will be rounded to the nearest unit.

- **ceilingStart**  
  Logical instruction to apply `ceiling_date` to the `startdate` rather than `floor_date` when rounding.

- **ceilingEnd**  
  Logical instruction to apply `ceiling_date` to the `enddate` rather than `floor_date` when rounding.

- **includeEnd**  
  Logical specifying that records associated with `enddate` should be included.
Value

A subset of the incoming `mts` time series object. (A list with `meta` and `data` dataframes.)

Note

This function is deprecated as of `MazamaTimeSeries 0.2.15`. Please use `mts_setTimeAxis` to shorten or lengthen the time axis of an `mts` object.

See Also

- `mts_filterData`
- `mts_filterDate`
- `mts_filterMeta`

---

**Description**

A generalized metadata filter for `mts` objects to choose rows/cases where conditions are true. Multiple conditions are combined with `&` or separated by a comma. Only rows where the condition evaluates to TRUE are kept. Rows where the condition evaluates to NA are dropped.

If an empty `mts` object is passed in, it is immediately returned, allowing for multiple filtering steps to be piped together and only checking for an empty `mts` object at the end of the pipeline.

**Usage**

```r
mts_filterMeta(mts, ...)
```

**Arguments**

- `mts` : `mts` object.
- `...` : Logical predicates defined in terms of the variables in `mts$meta`.

**Value**

A subset of the incoming `mts` time series object. (A list with `meta` and `data` dataframes.)

**Note**

Filtering is done on variables in `mts$meta`.

**See Also**

- `mts_filterData`
- `mts_filterDate`
- `mts_filterDatetime`
**Examples**

```r
library(MazamaTimeSeries)

# Filter for all labels with "SCSH"
scap <-
  example_mts %>%
  mts_filterMeta(communityRegion == "El Monte")

dplyr::select(scap$meta, ID, label, longitude, latitude, communityRegion)

head(scap$data)
```

---

**mts_getDistance**

*Calculate distances from mts time series locations to a location of interest*

**Description**

This function uses the `geodist` package to return the distances (meters) between mts locations and a location of interest. These distances can be used to create a mask identifying monitors within a certain radius of the location of interest.

**Usage**

```r
mts_getDistance(
  mts = NULL,
  longitude = NULL,
  latitude = NULL,
  measure = c("geodesic", "haversine", "vincenty", "cheap")
)
```

**Arguments**

- `mts` _mts_ object.
- `longitude` Longitude of the location of interest.
- `latitude` Latitude of the location of interest.
- `measure` One of "geodesic", "haversine", "vincenty" or "cheap"

**Value**

Vector of of distances (meters) named by deviceDeploymentID.

**Note**

The measure "cheap" may be used to speed things up depending on the spatial scale being considered. Distances calculated with measure = "cheap" will vary by a few meters compared with those calculated using measure = "geodesic".
Examples

```r
library(MazamaTimeSeries)

# Garfield Medical Center in LA
longitude <- -118.12321
latitude <- 34.06775

distances <- mts_getDistance(
  mts = example_mts,
  longitude = longitude,
  latitude = latitude
)

# Which sensors are within 1000 meters of Garfield Med Ctr?
distances[distances <= 1000]
```

---

**mts_isEmpty**  
*Test for an empty mts object*

**Description**

Convenience function for `nrow(mts$data) == 0`. This makes for more readable code in functions that need to test for this.

**Usage**

```r
mts_isEmpty(mts)
```

**Arguments**

- `mts`  
  *mts object*

**Value**

TRUE if no data exist in `mts`, FALSE otherwise.

**Examples**

```r
library(MazamaTimeSeries)

mts_isEmpty(example_mts)
```
mts_isValid

Test mts object for correct structure

Description

The mts is checked for the presence of core meta and data columns.

Core meta columns include:

- `deviceDeploymentID` – unique identifier (see MazmaLocationUtils)
- `deviceID` – device identifier
- `locationID` – location identifier (see MazmaLocationUtils)
- `locationName` – English language name
- `longitude` – decimal degrees E
- `latitude` – decimal degrees N
- `elevation` – elevation of station in m
- `countryCode` – ISO 3166-1 alpha-2
- `timezone` – Olson time zone

Core data columns include:

- `datetime` – measurement time (UTC)

Usage

mts_isValid(mts = NULL, verbose = FALSE)

Arguments

- mts: mts object
- verbose: Logical specifying whether to produce detailed warning messages.

Value

Invisibly returns TRUE if mts has the correct structure, FALSE otherwise.

See Also

mts_check

Examples

library(MazamaTimeSeries)

print(mts_isValid(example_mts))
mts_sample  

Sample time series for an mts time series object

Description

Reduce the number of records (timesteps) in the data dataframe of the incoming mts through random sampling.

Usage

mts_sample(
  mts = NULL,
  sampleSize = 5000,
  seed = NULL,
  keepOutliers = FALSE,
  width = 5,
  thresholdMin = 3
)

Arguments

mts  
mts object.

sampleSize  
Non-negative integer giving the number of rows to choose.

seed  
Integer passed to set.seed for reproducible sampling.

keepOutliers  
Logical specifying a graphics focused sampling algorithm that retains outliers (see Details).

width  
Integer width of the rolling window used for outlier detection.

thresholdMin  
Numeric threshold for outlier detection.

Details

When keepOutliers = FALSE, random sampling is used to provide a statistically relevant subsample of the data.

Value

A subset of the given mts object.

An mts time series object with fewer timesteps. (A list with meta and data dataframes.)

Outlier Detection

When keepOutliers = TRUE, a customized sampling algorithm is used that attempts to create subsets for use in plotting that create plots that are visually identical to plots using all data. This is accomplished by preserving outliers and only sampling data in regions where overplotting is expected.

The process is as follows:
1. find outliers using MazamaRollUtils::findOutliers()
2. create a subset consisting of only outliers
3. sample the remaining data
4. merge the outliers and sampled data

This algorithm works best when the mts object has only one or two timeseries.

The width and thresholdMin parameters determine the number of outliers detected. For hourly data, a width of 5 and a thresholdMin of 3 or 4 seem to find many visually obvious outliers.

Users attempting to optimize plotting speed for lengthy time series are encouraged to experiment with these two parameters along with sampleSize and review the results visually.

See findOutliers.

mts_select
Reorder and subset time series within an mts time series object

Description
This function acts similarly to dplyr::select() working on mts$data. The returned mts object will contain only those time series identified by deviceDeploymentID in the order specified.

This can be used the specify a preferred order and is helpful when using faceted plot functions based on ggplot such as those found in the AirMonitorPlots package.

Usage
mts_select(mts = NULL, deviceDeploymentID = NULL)

Arguments
mts mts object.
deviceDeploymentID Vector of timeseries unique identifiers.

Value
A reordered (subset) of the incoming mts time series object. (A list with meta and data dataframes.)

See Also
mts_selectWhere
library(MazamaTimeSeries)

# Filter for "El Monte"
El_Monte <-
  example_mts %>%
  mts_filterMeta(communityRegion == "El Monte")

ids <- El_Monte$meta$deviceDeploymentID
rev_ids <- rev(ids)

print(ids)
print(rev_ids)

rev_El_Monte <-
  example_mts %>%
  mts_select(rev_ids)

print(rev_El_Monte$meta$deviceDeploymentID)

mts_selectWhere  Data-based subsetting of time series within an mts object.

Description

Subsetting of mts acts similarly to tidyselect::where() working on mts$data. The returned mts object will contain only those time series where FUN applied to the time series data returns TRUE.

Usage

mts_selectWhere(mts, FUN)

Arguments

mts mts object.
FUN A function applied to time series data that returns TRUE or FALSE.

Value

A subset of the incoming mts object. (A list with meta and data dataframes.)

See Also

mts_select
Examples

library(MazamaTimeSeries)

# Show all Camp_Fire locations
Camp_Fire$meta$locationName

# Set a threshold
threshold <- 500

# Find time series with data at or above this threshold
worst_sites <-
  Camp_Fire %>%
  mts_selectWhere(
    function(x) { any(x >= threshold, na.rm = TRUE) }
  )

# Show the worst locations
worst_sites$meta$locationName

mts_setTimeAxis

Extend/contract mts time series to new start and end times

Description

Extends or contracts the time range of an mts object by adding/removing time steps at the start and end and filling any new time steps with missing values. The resulting time axis is guaranteed to be a regular, hourly axis with no gaps using the same timezone as the incoming mts object. This is useful when you want to place separate mts objects on the same time axis for plating.

If either startdate or enddate is missing, the start or end of the timeseries in mts will be used.

Usage

mts_setTimeAxis(mts = NULL, startdate = NULL, enddate = NULL, timezone = NULL)

Arguments

mts mts object.
startdate Desired start date (ISO 8601).
enddate Desired end date (ISO 8601).
timezone Olson timezone used to interpret startdate and enddate.

Value

The incoming mts time series object defined on a new time axis. (A list with meta and data dataframes.)
Note

If startdate or enddate is a POSIXct value, then timezone will be set to the timezone associated with startdate or enddate. In this common case, you don’t need to specify timezone explicitly.

If neither startdate nor enddate is a POSIXct value AND no timezone is supplied, the timezone will be inferred from the most common timezone found in mts.

Examples

```r
library(MazamaTimeSeries)

# Default range
range(example_mts$data$datetime)

# One-sided extend with user specified timezone
example_mts %>%
  mts_setTimeAxis(enddate = 20190815, timezone = "UTC") %>%
  mts_extractData() %>%
  dplyr::pull(datetime) %>%
  range()

# Two-sided extend with user specified timezone
example_mts %>%
  mts_setTimeAxis(20190615, 20190815, timezone = "UTC") %>%
  mts_extractData() %>%
  dplyr::pull(datetime) %>%
  range()

# Two-sided extend without timezone (uses timezone from mts$meta$timezone)
example_mts %>%
  mts_setTimeAxis(20190615, 20190815) %>%
  mts_extractData() %>%
  dplyr::pull(datetime) %>%
  range()
```

mts_slice_head

Subset time series based on their position

Description

An mts object is reduced so as to contain only the first or last n timeseries. These functions work similarly to `dplyr::slice_head` and `dplyr::slice_tail` but apply to both dataframes in the mts object.

This is primarily useful when the mts object has been ordered by a previous call to `mts_arrange` or by some other means.

`slice_head()` selects the first and `slice_tail()` the last timeseries in the object.
mts_summarize

Usage

mts_slice_head(mts, n = 5)

mts_slice_tail(mts, n = 5)

Arguments

mts  mts object.
n  Number of rows of mts$meta to select.

Value

A subset of the incoming mts time series object. (A list with meta and data dataframes.)

Examples

library(MazamaTimeSeries)

# Find lowest elevation sites
Camp_Fire %>%
  mts_filterMeta(!is.na(elevation)) %>%
  mts_arrange(elevation) %>%
  mts_slice_head(n = 5) %>%
  mts_extractMeta() %>%
  dplyr::select(elevation, locationName)

# Find highest elevation sites
Camp_Fire %>%
  mts_filterMeta(!is.na(elevation)) %>%
  mts_arrange(elevation) %>%
  mts_slice_tail(n = 5) %>%
  mts_extractMeta() %>%
  dplyr::select(elevation, locationName)

mts_summarize  Create summary time series for an mts time series object

Description

Individual time series in mts$data are grouped by unit and then summarized using FUN.

The most typical use case is creating daily averages where each day begins at midnight. This function interprets times using the mts$data$datetime tzone attribute so be sure that is set properly.

Day boundaries are calculated using the specified timezone or, if NULL, the most common (hopefully only!) time zone found in mts$meta$timezone. Leaving timezone = NULL, the default, results in “local time” date filtering which is the most common use case.
Usage

mts_summarize(
  mts, 
  timezone = NULL, 
  unit = c("day", "week", "month", "year"), 
  FUN = NULL, 
  ..., 
  minCount = NULL
)

Arguments

mts     mts object.
timezone Olson timezone used to interpret dates.
unit     Unit used to summarize by (e.g. "day").
FUN     Function used to summarize time series.
...     Additional arguments to be passed to FUN (e.g. na.rm = TRUE).
minCount Minimum number of valid data records required to calculate summaries. Time periods with fewer valid records will be assigned NA.

Value

An mts time series object containing daily (or other) statistical summaries. (A list with meta and data dataframes.)

Note

Because the returned mts object is defined on a daily axis in a specific time zone, it is important that the incoming mts contain timeseries associated with a single time zone.

Examples

library(MazamaTimeSeries)

daily <-
  mts_summarize(
    mts = Carmel_Valley, 
    timezone = NULL, 
    unit = "day", 
    FUN = mean, 
    na.rm = TRUE, 
    minCount = 18
  )

# Daily means
head(daily$data)
mts_trim

Trim mts time series by removing missing values

Description
Trims the time range of an mts object by removing time steps from the start and end that contain only missing values.

Usage
mts_trim(mts = NULL)

Arguments
mts
mts object.

Value
A subset of the incoming mts time series object. (A list with meta and data dataframes.)

Examples
library(MazamaTimeSeries)

# Untrimmed range
range(example_mts$data$datetime)

# Replace the first 50 data values for all non-"datetime" columns
example_mts$data[1:50, -1] <- NA

# Trimmed range
mts_trimmed <- mts_trim(example_mts)
range(mts_trimmed$data$datetime)

mts_trimDate

Trim mts time series object to full days

Description
Trims the date range of an mts object to local time date boundaries which are within the time range of the mts object. This has the effect of removing partial-day data records at the start and end of the timeseries and is useful when calculating full-day statistics.

By default, multi-day periods of all-missing data at the beginning and end of the timeseries are removed before trimming to date boundaries. If trimEmptyDays = FALSE all records are retained except for partial days beyond the first and after the last date boundary.

Day boundaries are calculated using the specified timezone or, if NULL, mts$meta$timezone. Leaving timezone = NULL, the default, results in "local time" date filtering which is the most common use case.
Usage

mts_trimDate(mts = NULL, timezone = NULL, trimEmptyDays = TRUE)

Arguments

mts  
mts object.

timezone  
Olson timezone used to interpret dates.

trimEmptyDays  
Logical specifying whether to remove days with no data at the beginning and end of the time range.

Value

A subset of the incoming mts time series object. (A list with meta and data dataframes.)

Examples

library(MazamaTimeSeries)

UTC_week <- mts_filterDate(
  example_mts,
  startdate = 20190703,
  enddate = 20190706,
  timezone = "UTC"
)

# UTC day boundaries
range(UTC_week$data$datetime)

# Trim to local time day boundaries
local_week <- mts_trimDate(UTC_week)
range(local_week$data$datetime)

---

requiredMetaNames  
Required columns for the 'meta' dataframe

Description

The 'meta' dataframe found in sts and mts objects is required to have a minimum set of information for proper functioning of the package. The names of these columns are specified in requiredMetaNames and include:

- deviceDeploymentID – unique identifier (see MazmaLocationUtils)
- deviceID – device identifier
- locationID – location identifier (see MazmaLocationUtils)
- locationName – English language name
sts_check

- longitude – decimal degrees E
- latitude – decimal degrees N
- elevation – elevation of station in m
- countryCode – ISO 3166-1 alpha-2
- timezone – Olson time zone

Usage

requiredMetaNames

Format

A vector with 10 elements

Details

requiredMetaNames

sts_check

Check sts object for validity

Description

Checks on the validity of an sts object. If any test fails, this function will stop with a warning message.

Usage

sts_check(sts)

Arguments

sts

sts object.

Value

Returns TRUE invisibly if the sts object is valid.

See Also

sts_isValid
Examples

```r
library(MazamaTimeSeries)

sts_check(example_sts)

# This would throw an error
if ( FALSE ) {
  broken_sts <- example_sts
  names(broken_sts) <- c('meta', 'bop')
  sts_check(broken_sts)
}
```

---

**sts_combine**  
*Combine multiple sts time series objects*

**Description**

Create a merged timeseries using of any number of *sts* objects for a single sensor. If *sts* objects are non-contiguous, the resulting *sts* will have gaps.

An error is generated if the incoming *sts* objects have non-identical `deviceDeploymentIDs`.

**Usage**

```r
sts_combine(..., replaceMeta = FALSE)
```

**Arguments**

- `...` Any number of valid `SingleTimeSeries` *sts* objects associated with a single `deviceDeploymentID`.
- `replaceMeta` Logical specifying whether to allow replacement of metadata.

**Value**

A `SingleTimeSeries` *sts* time series object containing records from all incoming *sts* time series objects. (A list with `meta` and `data` dataframes.)

**Note**

Data are combined with a "later is better" sensibility where any data overlaps exist. To handle this, incoming *sts* objects are first split into "shared" and "unshared" parts.

Any "shared" parts are ordered based on the time stamp of their last record. Then `dplyr::distinct()` is used to remove records with duplicate `datetime` fields. Any data records found in "later" *sts* objects are preferentially retained before the "shared" data are finally reordered by ascending `datetime`.

The final step is combining the "shared" and "unshared" parts.
Examples

```r
library(MazamaTimeSeries)

aug01_08 <-
example_sts %>%
sts_filterDate(20180801, 20180808)

aug15_22 <-
example_sts %>%
sts_filterDate(20180815, 20180822)

aug01_22 <- sts_combine(aug01_08, aug15_22)

plot(aug01_22$data$datetime)
```

### Description

Three successive steps are used to guarantee that the datetime axis contains no repeated values:

1. remove any duplicate records
2. guarantee that rows are in datetime order
3. average together fields for any remaining records that share the same datetime

### Usage

```r
sts_distinct(sts)
```

### Arguments

- `sts`  
  sts object

### Value

An `sts` object where each record is associated with a unique time. (A list with meta and data dataframes.)
**sts_extractDataFrame**  
*Extract dataframes from sts objects*

**Description**
These functions are convenient wrappers for extracting the dataframes that comprise a *sts* object. These functions are designed to be useful when manipulating data in a pipeline using %>%.

Below is a table showing equivalent operations for each function.
sts_extractData(sts) is equivalent to sts$data.
sts_extractMeta(sts) is equivalent to sts$meta.

**Usage**
sts_extractData(sts)
sts_extractMeta(sts)

**Arguments**
**sts**  
*sts* object to extract dataframe from.

**Value**
A dataframe from the *sts* object.

---

**sts_filter**  
*General purpose data filtering for sts time series objects*

**Description**
A generalized data filter for *sts* objects to choose rows/cases where conditions are true. Multiple conditions are combined with & or separated by a comma. Only rows where the condition evaluates to TRUE are kept. Rows where the condition evaluates to NA are dropped.

If an empty *sts* object is passed in, it is immediately returned, allowing for multiple filtering steps to be piped together and only checking for an empty *sts* object at the end of the pipeline.

**Usage**
sts_filter(sts, ...)

**Arguments**
**sts**  
*sts* object.

**...**  
Logical predicates defined in terms of the variables in *sts*$data.
sts_filterDate

Value

A subset of the incoming sts time series object. (A list with meta and data dataframes.)

Note

Filtering is done on values in sts$data.

See Also

sts_filterDate
sts_filterDatetime

Examples

library(MazamaTimeSeries)

unhealthy <- sts_filter(example_sts, pm25_A > 55.5, pm25_B > 55.5)
head(unhealthy$data)

Description

Subsets a MazamaSingleTimeseries object by date. This function always filters to day-boundaries. For sub-day filtering, use sts_filterDatetime().

Dates can be anything that is understood by MazamaCoreUtils::parseDatetime() including either of the following recommended formats:

- "YYYYmmdd"
- "YYYY-mm-dd"

Timezone determination precedence assumes that if you are passing in POSIXct values then you know what you are doing.

1. get timezone from startdate if it is POSIXct
2. use passed in timezone
3. get timezone from sts
Usage

sts_filterDate(
  sts = NULL,
  startdate = NULL,
  enddate = NULL,
  timezone = NULL,
  unit = "sec",
  ceilingStart = FALSE,
  ceilingEnd = FALSE
)

Arguments

sts  MazamaSingleTimeseries sts object.
startdate  Desired start datetime (ISO 8601).
enddate  Desired end datetime (ISO 8601).
timezone  Olson timezone used to interpret dates.
unit  Units used to determine time at end-of-day.
ceilingStart  Logical instruction to apply ceiling_date to the startdate rather than floor_date
ceilingEnd  Logical instruction to apply ceiling_date to the enddate rather than floor_date

Value

A subset of the incoming sts time series object. (A list with meta and data dataframes.)

Note

The returned data will run from the beginning of startdate until the beginning of enddate – i.e. no values associated with enddate will be returned. The exception being when enddate is less than 24 hours after startdate. In that case, a single day is returned.

See Also

sts_filter
sts_filterDatetime

Examples

library(MazamaTimeSeries)

example_sts %>%
  sts_filterDate(startdate = 20180808, enddate = 20180815) %>%
  sts_extractData() %>%
  head()
sts_filterDatetime

Datetime filtering for sts time series objects

Description

Subsets a MazamaSingleTimeseries object by datetime. This function allows for sub-day filtering as opposed to sts_filterDate() which always filters to day-boundaries.

Datetimes can be anything that is understood by MazamaCoreUtils::parseDatetime(). For non-POSIXct values, the recommended format is "YYYY-mm-dd HH:MM:SS".

Timezone determination precedence assumes that if you are passing in POSIXct values then you know what you are doing.

1. get timezone from startdate if it is POSIXct
2. use passed in timezone
3. get timezone from sts

Usage

sts_filterDatetime(
  sts = NULL,
  startdate = NULL,
  enddate = NULL,
  timezone = NULL,
  unit = "sec",
  ceilingStart = FALSE,
  ceilingEnd = FALSE,
  includeEnd = FALSE
)

Arguments

sts  MazamaSingleTimeseries sts object.
startdate  Desired start datetime (ISO 8601).
enddate  Desired end datetime (ISO 8601).
timezone  Olson timezone used to interpret dates.
unit  Units used to determine time at end-of-day.
ceilingStart  Logical instruction to apply ceiling_date to the startdate rather than floor_date
ceilingEnd  Logical instruction to apply ceiling_date to the enddate rather than floor_date
includeEnd  Logical specifying that records associated with enddate should be included.

Value

A subset of the incoming sts time series object. (A list with meta and data dataframes.)
Note

The returned sts object will contain data running from the beginning of startdate until the beginning of enddate – i.e. no values associated with enddate will be returned. To include enddate you can specify includeEnd = TRUE.

See Also

sts_filter
sts_filterDate

Examples

library(MazamaTimeSeries)

example_sts %>%
  stst_filterDatetime(
    startdate = "2018-08-08 06:00:00",
    enddate = "2018-08-14 18:00:00"
  ) %>%
  stst_extractData() %>%
  head()

---

sts_isEmpty Test for empty sts object

Description

Convenience function for nrow(sts$data) == 0. This makes for more readable code in functions that need to test for this.

Usage

sts_isEmpty(sts)

Arguments

sts sts object

Value

TRUE if no data exist in sts, FALSE otherwise.

Examples

library(MazamaTimeSeries)

sts_isEmpty(example_sts)
**Description**

The `sts` is checked for the presence of core meta and data columns. Core meta columns include:

- `deviceDeploymentID` – unique identifier (see `MazmaLocationUtils`)
- `deviceID` – device identifier
- `locationID` – location identifier (see `MazmaLocationUtils`)
- `locationName` – English language name
- `longitude` – decimal degrees E
- `latitude` – decimal degrees N
- `elevation` – elevation of station in m
- `countryCode` – ISO 3166-1 alpha-2
- `timezone` – Olson time zone

Core data columns include:

- `datetime` – measurement time (UTC)

**Usage**

```r
sts_isValid(sts = NULL, verbose = FALSE)
```

**Arguments**

- `sts` — `sts` object
- `verbose` — Logical specifying whether to produce detailed warning messages.

**Value**

`TRUE` if `sts` has the correct structure, `FALSE` otherwise.

**Examples**

```r
library(MazamaTimeSeries)

sts_isValid(example_sts)
```
sts_summarize

Create summary time series for an sts time series object

Description

Columns of numeric data in \texttt{sts$data} are grouped by \texttt{unit} and then summarized using \texttt{FUN}.

Columns with non-numeric data are summarized by just picking the first occurrence in each \texttt{unit}.

This preserves the utility of columns containing repeated metadata.

The most typical use case is creating daily averages where each day begins at midnight. Day boundaries are calculated using the specified \texttt{timezone} or, if \texttt{NULL}, the time zone found in \texttt{sts$meta$timezone[1]}. Leaving \texttt{timezone = NULL}, the default, results in "local time" date filtering which is the most common use case.

Usage

\begin{verbatim}
sts_summarize(
    sts,          
    timezone = NULL, 
    unit = c("day", "week", "month", "year"), 
    FUN = NULL, 
    ..., 
    minCount = NULL
)
\end{verbatim}

Arguments

\begin{itemize}
\item \texttt{sts} \textit{sts} object.
\item \texttt{timezone} Olson timezone used to interpret dates.
\item \texttt{unit} Unit used to summarize by \textit{(e.g.} "day"\textit{)}.
\item \texttt{FUN} Function used to summarize time series.
\item \texttt{...} Additional arguments to be passed to \texttt{FUN} \textit{(e.g.} \texttt{na.rm = TRUE}\textit{)}.
\item \texttt{minCount} Minimum number of valid data records required to calculate summaries. Time periods with fewer valid records will be assigned \texttt{NA}.
\end{itemize}

Value

An \texttt{sts} time series object containing daily (or other) statistical summaries. (A list with \texttt{meta} and \texttt{data} dataframes.)
sts_trimDate

Trims the date range of a *sts* object to local time date boundaries which are *within* the range of data. This has the effect of removing partial-day data records at the start and end of the timeseries and is useful when calculating full-day statistics.

Day boundaries are calculated using the specified timezone or, if NULL, from *sts$meta$timezone*.

**Usage**

```r
sts_trimDate(sts = NULL, timezone = NULL)
```

**Arguments**

- **sts**: SingleTimeSeries *sts* object.
- **timezone**: Olson timezone used to interpret dates.

**Value**

A subset of the incoming *sts* time series object. (A list with meta and data dataframes.)

**Examples**

```r
library(MazamaTimeSeries)

UTC_week <- sts_filterDate(
    example_sts,
    startdate = 20180808,
    enddate = 20180815,
    timezone = "UTC"
)

# UTC day boundaries
head(UTC_week$data)

# Trim to local time day boundaries
local_week <- sts_trimDate(UTC_week)
head(local_week$data)
```
timeInfo

Get time related information

Description

Calculate the local time at the target location, as well as sunrise, sunset and solar noon times, and create several temporal masks.

The returned dataframe will have as many rows as the length of the incoming UTC time vector and will contain the following columns:

- `localStdTime_UTC` – UTC representation of local `standard` time
- `daylightSavings` – logical mask = TRUE if daylight savings is in effect
- `localTime` – local clock time
- `sunrise` – time of sunrise on each localTime day
- `sunset` – time of sunset on each localTime day
- `solarnoon` – time of solar noon on each localTime day
- `day` – logical mask = TRUE between sunrise and sunset
- `morning` – logical mask = TRUE between sunrise and solarnoon
- `afternoon` – logical mask = TRUE between solarnoon and sunset
- `night` – logical mask = opposite of day

Usage

timeInfo(time = NULL, longitude = NULL, latitude = NULL, timezone = NULL)

Arguments

time POSIXct vector with specified timezone,
longitude Longitude of the location of interest.
latitude Latitude of the location of interest.
timezone Olson timezone at the location of interest.

Details

NOAA used the reference below to develop their Sunrise/Sunset
https://gml.noaa.gov/grad/solcalc/sunrise.html and Solar Position
https://gml.noaa.gov/grad/solcalc/azel.html Calculators. The algorithms include corrections for atmospheric refraction effects.

Input can consist of one location and at least one POSIXct times, or one POSIXct time and at least one location. solarDep is recycled as needed.

Do not use the daylight savings time zone string for supplying `dateTime`, as many OS will not be able to properly set it to standard time when needed.

The `localStdTime_UTC` column in the returned dataframe is primarily for internal use and provides an important tool for creating LST daily averages and LST axis labeling.
Value

A dataframe with times and masks.

Attribution

Internal functions used for ephemerides calculations were copied verbatim from the now deprecated `maptools` package source code in an effort to reduce the number of package dependencies.

Warning

Compared to NOAA's original Javascript code, the sunrise and sunset estimates from this translation may differ by +/- 1 minute, based on tests using selected locations spanning the globe. This translation does not include calculation of prior or next sunrises/sunsets for locations above the Arctic Circle or below the Antarctic Circle.

Local Standard Time

US EPA regulations mandate that daily averages be calculated based on "Local Standard Time" (LST) (i.e. never shifting to daylight savings). To ease work in a regulatory context, LST times are included in the returned dataframe.

References


Note

NOAA notes that “for latitudes greater than 72 degrees N and S, calculations are accurate to within 10 minutes. For latitudes less than +/- 72 degrees accuracy is approximately one minute.”

Author(s)

Sebastian P. Luque <spluque@gmail.com>, translated from Greg Pelletier's <gpel461@ecy.wa.gov>

VBA code (available from https://ecology.wa.gov/Research-Data/Data-resources/Models-spreadsheets/Modeling-the-environment/Models-tools-for-TMDLs), who in turn translated it from original Javascript code by NOAA (see Details). Roger Bivand <roger.bivand@nhh.no> adapted the code to work with sp classes. Jonathan Callahan <jonathan.callahan@gmail.com> adapted the source code from the `maptools` package to work with `MazamaTimeSeries` classes.

Examples

```r
library(MazamaTimeSeries)

Carmel <-
  Carmel_Valley %>%
  mts_filterDate(20160801, 20160810)

# Create timeInfo object for this monitor
ti <- timeInfo(
  Carmel$data$data$datetime,
```
Carmel$meta$longitude,
Carmel$meta$latitude,
Carmel$meta$timezone
}

t(ti[6:9,])

# Subset the data based on day/night masks
data_day <- Carmel$data[ti$day,]
data_night <- Carmel$data[ti$night,]

# Build two monitor objects
Carmel_day <- list(meta = Carmel$meta, data = data_day)
Carmel_night <- list(meta = Carmel$meta, data = data_night)

# Plot them
plot(Carmel_day$data, pch = 8, col = 'goldenrod')
points(Carmel_night$data, pch = 16, col = 'darkblue')
Index

* datasets
  Camp_Fire, 3
  Carmel_Valley, 4
  example_mts, 4
  example_raws, 5
  example_sts, 6
  requiredMetaNames, 28

  arrange, 7
  Camp_Fire, 3
  Carmel_Valley, 4
  ceiling_date, 14, 15, 34, 35

  dplyr::slice_head, 24
  dplyr::slice_tail, 24

  example_mts, 4
  example_raws, 5
  example_sts, 6

  findOutliers, 21
  floor_date, 14, 15, 34, 35

  MazamaTimeSeries, 6
  MazamaTimeSeries-package
    (MazamaTimeSeries), 6

  mts_arrange, 7, 24
  mts_check, 7, 19
  mts-collapse, 8
  mts_combine, 10
  mts_distinct, 11
  mts_extractData (mts_extractDataFrame),
    12
  mts_extractDataFrame, 12
  mts_extractMeta (mts_extractDataFrame),
    12
  mts_filterData, 12, 14, 16
  mts_filterDate, 13, 13, 16
  mts_filterDatetime, 13, 14, 15, 16
  mts_filterMeta, 13, 14, 16, 16

  mts_getDistance, 17
  mts_isEmpty, 18
  mts_isValid, 8, 19
  mts_sample, 20
  mts_select, 21, 22
  mts_selectWhere, 21, 22
  mts_setTimeAxis, 15, 16, 23
  mts_slice_head, 24
  mts_slice_tail (mts_slice_head), 24
  mts_summarize, 25
  mts_trim, 27
  mts_trimDate, 27

  requiredMetaNames, 28

  set.seed, 20
  sts_check, 29
  sts_combine, 30
  sts_distinct, 31
  sts_extractData (sts_extractDataFrame),
    32
  sts_extractDataFrame, 32
  sts_extractMeta (sts_extractDataFrame),
    32
  sts_filter, 32, 34, 36
  sts_filterDate, 33, 33, 36
  sts_filterDatetime, 33, 34, 35
  sts_isEmpty, 36
  sts_isValid, 29, 37
  sts_summarize, 38
  sts_trimDate, 39

  timeInfo, 40