Package 'MazamaCoreUtils'

July 20, 2020

Type Package

Version 0.4.4

Title Utility Functions for Production R Code

Author Jonathan Callahan [aut, cre],
Eli Grosman [aut],
Spencer Pease [aut],
Thomas Bergamaschi [aut]

Maintainer Jonathan Callahan <jonathan.s.callahan@gmail.com>

Description A suite of utility functions providing functionality commonly needed for production level projects such as logging, error handling, and cache management.

License GPL-3

URL https://github.com/MazamaScience/MazamaCoreUtils

BugReports https://github.com/MazamaScience/MazamaCoreUtils/issues

Depends R (>= 3.1.0)

Imports devtools, dplyr, futile.logger, lubridate, stringr, magrittr, purrr, tibble, rlang (>= 0.1.2), xml2, rvest

Suggests knitr, markdown, testthat (>= 2.1.0), rmarkdown, roxygen2

Encoding UTF-8

VignetteBuilder knitr

LazyData true

RoxygenNote 7.1.1

NeedsCompilation no

Repository CRAN

Date/Publication 2020-07-20 21:10:03 UTC

R topics documented:

`dateRange` *Create a POSIXct date range*

Description

Uses incoming parameters to return a pair of `POSIXct` times in the proper order. The first returned time will be midnight of the desired starting date. The second returned time will represent the "end of the day" of the requested or calculated `enddate` boundary.

Note that the returned end date will be one `unit` prior to the start of the requested `enddate` unless `ceilingEnd = TRUE` in which case the entire `enddate` will be included up to the last `unit`.

The `ceilingEnd` argument addresses the ambiguity of a phrase like: "August 1-8". With `ceilingEnd = FALSE` (default) this pharse means "through the beginning of Aug 8". With `ceilingEnd = TRUE` it means "through the end of Aug 8".

So, to get 24 hours of data staring on Jan 01, 2019 you would specify:

```
> MazamaCoreUtils::dateRange(20190101, 20190102, timezone = "UTC")
[1] "2019-01-01 00:00:00 UTC" "2019-01-01 23:59:59 UTC"
```

or

```
> MazamaCoreUtils::dateRange(20190101, 20190101,
                             timezone = "UTC", ceilingEnd = TRUE)
[1] "2019-01-01 00:00:00 UTC" "2019-01-01 23:59:59 UTC"
```

The required `timezone` parameter must be one of those found in `OlsonNames`.

Dates can be anything that is understood by `lubrdiate::parse_date_time()` using the `Ymd[HMS]` orders. This includes:

- `"YYYYmmdd"`
- `"YYYYmmddHHMMSS"`
- `"YYYY-mm-dd"`
- `"YYYY-mm-dd H"`
- `"YYYY-mm-dd H:M"`
- `"YYYY-mm-dd H:M:S"`

Usage

```
dateRange(
  startdate = NULL,
  enddate = NULL,
  timezone = NULL,
  unit = "sec",
  ceilingStart = FALSE,
  ceilingEnd = FALSE,
  days = 7
)
```

Arguments

`startdate`	Desired start datetime (ISO 8601).
`enddate`	Desired end datetime (ISO 8601).
`timezone`	Olson timezone used to interpret dates (required).
`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`
`days`	Number of days of data to include.

Value

A vector of two `POSIXcts`.

Default Arguments

In the case when either `startdate` or `enddate` is missing, it is created from the non-missing values plus/minus `days`. If both `startdate` and `enddate` are misssing, `enddate` is set to `now` (with the given `timezone`), and then `startdate` is calculated using `enddate -days`.

End-of-Day Units

The second of the returned `POSIXcts` will end one `unit` before the specified `enddate`. Acceptable units are `"day"`,`"hour"`,`"min"`,`"sec"`.

The aim is to quickly calculate full-day date ranges for time series whose values are binned at different units. Thus, if `unit = "min"`, the returned value associated with `enddate` will always be at 23:59:00 in the requested time zone.

POSIXct inputs

When `startdate` or `enddate` are already `POSIXct` values, they are converted to the timezone specified by `timezone` without altering the physical instant in time the input represents. This is different from the behavior of `parse_date_time` (which powers this function), which will force `POSIXct` inputs into a new timezone, altering the physical moment of time the input represents.

Parameter precedence

It is possible to supply input paramters that are in conflict. For example:

`dateRange("2019-01-01","2019-01-08",days = 3,timezone = "UTC")`

The `startdate` and `enddate` parameters would imply a 7-day range which is in conflict with `days = 3`. The following rules resolve conflicts of this nature:

1. When `startdate` and `enddate` are both specified, the `days` parameter is ignored.
2. When `startdate` is missing, `ceilingStart` is ignored and the first returned time will depend on the combination of `enddate`, `days` and `ceilingEnd`.
3. When `enddate` is missing, `ceilingEnd` is ignored and the second returned time depends on `ceilingStart` and `days`.

Examples

```
dateRange("2019-01-08", timezone = "UTC")
dateRange("2019-01-08", unit = "min", timezone = "UTC")
dateRange("2019-01-08", unit = "hour", timezone = "UTC")
dateRange("2019-01-08", unit = "day", timezone = "UTC")
dateRange("2019-01-08", "2019-01-11", timezone = "UTC")
dateRange(enddate = 20190112, days = 3,
          unit = "day", timezone = "America/Los_Angeles")
```

dateSequence *Create a POSIXct date sequence*

Description

Uses incoming parameters to return a seqeunce of POSIXct times at local midnight in the specified timezone. The first returned time will be midnight of the requested startdate. The final returned time will be midnight (*at the beginning*) of the requested enddate.

The ceilingEnd argument addresses the ambiguity of a phrase like: "August 1-8". With ceilingEnd = FALSE (default) this pharse means "through the beginning of Aug 8". With ceilingEnd = TRUE it means "through the end of Aug 8".

The required timezone parameter must be one of those found in OlsonNames.

Dates can be anything that is understood by lubrdiate::parse_date_time() using the Ymd[HMS] orders. This includes:

- "YYYYmmdd"
- "YYYYmmddHHMMSS"
- "YYYY-mm-dd"
- "YYYY-mm-dd H"
- "YYYY-mm-dd H:M"
- "YYYY-mm-dd H:M:S"

All hour-minute-second information is removed after parsing.

Usage

```
dateSequence(
  startdate = NULL,
  enddate = NULL,
  timezone = NULL,
  ceilingEnd = FALSE
)
```

Arguments

startdate	Desired start datetime (ISO 8601).
enddate	Desired end datetime (ISO 8601).
timezone	Olson timezone used to interpret dates (required).
ceilingEnd	Logical instruction to apply ceiling_date to the enddate rather than floor_date

Value

A vector of POSIXcts at midnight local time.

POSIXct inputs

When `startdate` or `enddate` are already `POSIXct` values, they are converted to the timezone specified by `timezone` without altering the physical instant in time the input represents. Only after conversion are they floored to midnight local time

Note

The main utility of this function is that it respects "clock time" and returns times associated with midnight regardless of daylight savings. This is in contrast to 'seq.Date(from, to, by = "day")' which creates a sequence of datetimes always separated by 24 hours.

Examples

```
dateSequence("2019-11-01", "2019-11-08", timezone = "America/Los_Angeles")
dateSequence("2019-11-01", "2019-11-07", timezone = "America/Los_Angeles",
             ceilingEnd = TRUE)

# Observe the handling of daylight savings
datetime <- dateSequence("2019-11-01", "2019-11-08",
                         timezone = "America/Los_Angeles")

datetime
lubridate::with_tz(datetime, "UTC")

# Passing in POSIXct values preserves the instant in time before flooring --
#   midnight Tokyo time is the day before in UTC
jst <- dateSequence(20190307, 20190315, timezone = "Asia/Tokyo")
jst
dateSequence(jst[1], jst[7], timezone = "UTC")
```

`html_getLinks` *Find all links in an html page*

Description

Parses an html page to extract all `<a href="...">...</a>` links and return them in a dataframe where `linkName` is the human readable name and `linkUrl` is the `href` portion. By default this function will return relative URLs.

This is especially useful for extracting data from an index page that shows the contents of a web accessible directory.

Wrapper functions `html_getLinkNames()` and `html_getLinkUrls()` return the appropriate columns as vectors.

Usage

```
html_getLinks(url = NULL, relative = TRUE)

html_getLinkNames(url = NULL)

html_getLinkUrls(url = NULL, relative = TRUE)
```

Arguments

url	URL or file path of an html page.
relative	Logical instruction to return relative URLs.

Value

A dataframe with `linkName` and/or `linkUrl` columns.

Examples

```
library(MazamaCoreUtils)

# US Census 2019 shapefiles
dataLinks <- html_getLinks("https://www2.census.gov/geo/tiger/GENZ2019/shp/")

dataLinks <- dataLinks %>%
  dplyr::filter(stringr::str_detect(linkName, "us_county"))
head(dataLinks, 10)
```

---

html_getTables *Find all tables in an html page*

Description

Parses an html page to extract all <table> elements and return them in a list of dataframes representing each table. The columns and rows of these dataframes are that of the table it represents. A single table can be extracted as a dataframe by passing the index of the table in addition to the url to `html_getTable()`.

Usage

```
html_getTables(url = NULL)

html_getTable(url = NULL, index = 1)
```

Arguments

url	URL or file path of an html page.
index	Index identifying which table to to return.

Value

A list of dataframes representing each table on a html page.

Examples

```
library(MazamaCoreUtils)

# Wikipedia's list of timezones
url <- "http://en.wikipedia.org/wiki/List_of_tz_database_time_zones"

# Extract tables
tables <- html_getTables(url)

# Extract the first table
# NOTE: Analogous to firstTable <- html_getTable(url, index = 1)
firstTable <- tables[[1]]

head(firstTable)
nrow(firstTable)
```

---

initializeLogging *Initialize standard log files*

Description

Convenience function that wraps logging initialization steps common to Mazama Science web services.

Usage

```
initializeLogging(logDir = NULL)
```

Arguments

logDir	Directory in which to write log files.

---

lintFunctionArgs *Lint a source file's function arguments*

Description

This function parses an R Script file, grouping function calls and the named arguments passed to those functions. Then, based on a set of rules, it is determined if functions of interest have specific named arguments specified.

Usage

```
lintFunctionArgs_file(filePath = NULL, rules = NULL, fullPath = FALSE)

lintFunctionArgs_dir(dirPath = "./R", rules = NULL, fullPath = FALSE)
```

Arguments

filePath	Path to a file, given as a length one character vector.
rules	A named list where the name of each element is a function name, and the value is a character vector of the named argument to check for. All arguments must be specified for a function to "pass".
fullPath	Logical specifying whether to display absolute paths.
dirPath	Path to a directory, given as a length one character vector.

Value

A tibble detailing the results of the lint.

Linting Output

The output of the function argument linter is a tibble with the following columns:

file_path path to the source file

line_number Line of the source file the function is on

column_number Column of the source file the function starts at

function_name The name of the function

named_args A vector of the named arguments passed to the function

includes_required True iff the function specifies all of the named arguments required by the given rules

Limitations

This function is only able to test for named arguments passed to a function. For example, it would report that `foo(x = bar, "baz")` has specified the named argument `x`, but not that `bar` was the value of the argument, or that `"baz"` had been passed as an unnamed argument.

Examples

```
## Not run:
# Example rule list for checking
exRules <- list(
  "fn_one" = "x",
  "fn_two" = c("foo", "bar")
)

# Example of using included timezone argument linter
lintFunctionArgs_file(
  "local_test/timezone_lint_test_script.R",
```

```
  MazamaCoreUtils::timezoneLintRules
)

## End(Not run)
```

`loadDataFile` *Load data from URL or local file*

Description

Loads pre-generated R binary files from a URL or a local directory. This function is intended to be called by other `~_load()` functions and can remove internet latencies when local versions of data are available.

For this reason, specification of `dataDir` always takes precedence over `dataUrl`.

Usage

```
loadDataFile(filename = NULL, dataUrl = NULL, dataDir = NULL)
```

Arguments

`filename`	Name of the data file to be loaded.
`dataUrl`	Remote URL directory for data files.
`dataDir`	Local directory containing data files.

Value

A data object.

`logger.debug` *Python-style logging statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `DEBUG` level log statements.

Usage

```
logger.debug(msg, ...)
```

Arguments

`msg`	Message with format strings applied to additional arguments.
`...`	Additional arguments to be formatted.

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(
  debugLog = "debug.log",
  infoLog = "info.log",
  errorLog = "error.log"
)

# But allow log statements at all levels within the code
logger.trace("trace statement #%d", 1)
logger.debug("debug statement")
logger.info("info statement %s %s", "with", "arguments")
logger.warn("warn statement %s", "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error("error message: %s", geterrmessage())
logger.fatal("fatal statement %s", "THE END")

## End(Not run)
```

---

logger.error *Python-style logging statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `ERROR` level log statements.

Usage

```
logger.error(msg, ...)
```

Arguments

`msg`	Message with format strings applied to additional arguments.
`...`	Additional arguments to be formatted.

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(
  debugLog = "debug.log",
  infoLog = "info.log",
  errorLog = "error.log"
)

# But allow log statements at all levels within the code
logger.trace("trace statement #%d", 1)
logger.debug("debug statement")
logger.info("info statement %s %s", "with", "arguments")
logger.warn("warn statement %s", "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error("error message: %s", geterrmessage())
logger.fatal("fatal statement %s", "THE END")

## End(Not run)
```

---

logger.fatal *Python-style logging statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `FATAL` level log statements.

Usage

```
logger.fatal(msg, ...)
```

Arguments

`msg`	Message with format strings applied to additional arguments.
`...`	Additional arguments to be formatted.

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(
  debugLog = "debug.log",
  infoLog = "info.log",
  errorLog = "error.log"
)

# But allow log statements at all levels within the code
logger.trace("trace statement #%d", 1)
logger.debug("debug statement")
logger.info("info statement %s %s", "with", "arguments")
logger.warn("warn statement %s", "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error("error message: %s", geterrmessage())
logger.fatal("fatal statement %s", "THE END")

## End(Not run)
```

---

logger.info *Python-style logging statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `INFO` level log statements.

Usage

```
logger.info(msg, ...)
```

Arguments

`msg`	Message with format strings applied to additional arguments.
`...`	Additional arguments to be formatted.

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(
  debugLog = "debug.log",
  infoLog = "info.log",
  errorLog = "error.log"
)

# But allow log statements at all levels within the code
logger.trace("trace statement #%d", 1)
logger.debug("debug statement")
logger.info("info statement %s %s", "with", "arguments")
logger.warn("warn statement %s", "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error("error message: %s", geterrmessage())
logger.fatal("fatal statement %s", "THE END")

## End(Not run)
```

---

logger.isInitialized *Check for initialization loggers*

Description

Returns TRUE if logging has been initialized. This allows packages to emit logging statements only if logging has already been set up, potentially avoiding 'futile.log' errors.

Usage

```
logger.isInitialized()
```

Value

TRUE if logging has already been initialized.

See Also

logger.setup

initializeLogging

Examples

```
## Not run:
logger.isInitialized()
logger.setup()
logger.isInitialized()

## End(Not run)
```

`logger.setLevel` *Set console log level*

Description

By default, the logger threshold is set to `FATAL` so that the console will typically receive no log messages. By setting the level to one of the other log levels: `TRACE,DEBUG,INFO,WARN,ERROR` users can see logging messages while running commands at the command line.

Usage

```
logger.setLevel(level)
```

Arguments

`level`	Threshold level.

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Set up console logging only
logger.setup()
logger.setLevel(DEBUG)

## End(Not run)
```

`logger.setup` *Set up python-style logging*

Description

Good logging allows package developers and users to create log files at different levels to track and debug lengthy or complex calculations. "Python-style" logging is intended to suggest that users should set up multiple log files for different log severities so that the `errorLog` will contain only log messages at or above the `ERROR` level while a `debugLog` will contain log messages at the `DEBUG` level as well as all higher levels.

Python-style log files are set up with `logger.setup()`. Logs can be set up for any combination of log levels. Accepting the default `NULL` setting for any log file simply means that log file will not be created.

Python-style logging requires the use of `logger.debug()` style logging statements as seen in the example below.

Usage

```
logger.setup(
  traceLog = NULL,
  debugLog = NULL,
  infoLog = NULL,
  warnLog = NULL,
  errorLog = NULL,
  fatalLog = NULL
)
```

Arguments

`traceLog`	File name or full path where `logger.trace()` messages will be sent.
`debugLog`	File name or full path where `logger.debug()` messages will be sent.
`infoLog`	File name or full path where `logger.info()` messages will be sent.
`warnLog`	File name or full path where `logger.warn()` messages will be sent.
`errorLog`	File name or full path where `logger.error()` messages will be sent.
`fatalLog`	File name or full path where `logger.fatal()` messages will be sent.

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.trace logger.debug logger.info logger.warn logger.error logger.fatal

Examples

```
## Not run:
# Only save three log files
logger.setup(
  debugLog = "debug.log",
  infoLog = "info.log",
  errorLog = "error.log"
)

# But allow lot statements at all levels within the code
logger.trace("trace statement #%d", 1)
logger.debug("debug statement")
logger.info("info statement %s %s", "with", "arguments")
logger.warn("warn statement %s", "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error("error message: %s", geterrmessage())
logger.fatal("fatal statement %s", "THE END")

## End(Not run)
```

---

logger.trace *Python-style logging statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate TRACE level log statements.

Usage

```
logger.trace(msg, ...)
```

Arguments

msg	Message with format strings applied to additional arguments.
...	Additional arguments to be formatted.

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(
  debugLog = "debug.log",
  infoLog = "info.log",
  errorLog = "error.log"
)

# But allow log statements at all levels within the code
logger.trace("trace statement #%d", 1)
logger.debug("debug statement")
logger.info("info statement %s %s", "with", "arguments")
logger.warn("warn statement %s", "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error("error message: %s", geterrmessage())
logger.fatal("fatal statement %s", "THE END")

## End(Not run)
```

---

logger.warn *Python-style logging statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `WARN` level log statements.

Usage

```
logger.warn(msg, ...)
```

Arguments

`msg`	Message with format strings applied to additional arguments.
`...`	Additional arguments to be formatted.

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(
  debugLog = "debug.log",
  infoLog = "info.log",
  errorLog = "error.log"
)

# But allow log statements at all levels within the code
logger.trace("trace statement #%d", 1)
logger.debug("debug statement")
logger.info("info statement %s %s", "with", "arguments")
logger.warn("warn statement %s", "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error("error message: %s", geterrmessage())
logger.fatal("fatal statement %s", "THE END")

## End(Not run)
```

---

logLevels *Log levels*

Description

Log levels matching those found in **futile.logger**. Available levels include:

FATAL ERROR WARN INFO DEBUG TRACE

Usage

```
FATAL
```

Format

An object of class integer of length 1.

manageCache *Manage the size of a cache*

Description

If cacheDir takes up more than maxCacheSize megabytes on disk, files will be removed in order of access time by default. Only files matching extensions are eligible for removal. Files can also be removed in order of change time with sortBy='ctime' or modification time with sortBy='mtime'.

The maxFileAge parameter can also be used to remove files that haven't been modified in a certain number of days. Fractional days are allowed. This removal happens without regard to the size of the cache and is useful for removing out-of-date data.

It is important to understand precisely what these timestamps represent:

- atime – File access time: updated whenever a file is opened.
- ctime – File change time: updated whenever a file's metadata changes e.g. name, permission, ownership.
- mtime – file modification time: updated whenever a file's contents change.

Usage

```
manageCache(
  cacheDir = NULL,
  extensions = c("html", "json", "pdf", "png"),
  maxCacheSize = 100,
  sortBy = "atime",
  maxFileAge = NULL
)
```

Arguments

cacheDir	Location of cache directory.
extensions	Vector of file extensions eligible for removal.
maxCacheSize	Maximum cache size in megabytes.
sortBy	Timestamp to sort by when sorting files eligible for removal. One of atime|ctime|mtime.
maxFileAge	Maximum age in days of files allowed in the cache.

Value

Invisibly returns the number of files removed.

Examples

```
# Create a cache directory and fill it with 1.6 MB of data
CACHE_DIR <- tempdir()
write.csv(matrix(1,400,500), file=file.path(CACHE_DIR,'m1.csv'))
write.csv(matrix(2,400,500), file=file.path(CACHE_DIR,'m2.csv'))
write.csv(matrix(3,400,500), file=file.path(CACHE_DIR,'m3.csv'))
write.csv(matrix(4,400,500), file=file.path(CACHE_DIR,'m4.csv'))
for (file in list.files(CACHE_DIR, full.names=TRUE)) {
  print(file.info(file)[,c(1,6)])
}

# Remove files based on access time until we get under 1 MB
manageCache(CACHE_DIR, extensions='csv', maxCacheSize=1, sortBy='atime')
for (file in list.files(CACHE_DIR, full.names=TRUE)) {
  print(file.info(file)[,c(1,6)])
}

# Or remove files based on modification time
manageCache(CACHE_DIR, extensions='csv', maxCacheSize=1, sortBy='mtime')
for (file in list.files(CACHE_DIR, full.names=TRUE)) {
  print(file.info(file)[,c(1,6)])
}
```

MazamaCoreUtils *Utility Functions for Production R Code*

Description

The MazamaCoreUtils package was created by MazamaScience to regularize our work building R-based web services.

The main goal of this package is to create an internally standardized set of functions that we can use in various systems that are being run operationally. Areas of functionality supported by this package include:

- python style logging
- simple error messaging
- cache management
- date parsing
- source code linting

packageCheck *Run package checks*

Description

When multiple developers are working on a package, it is crucially important that they check their code changes *often*. After merging changes from multiple developers it is equally important to check the package *thoroughly*.

The problem is that frequent checks should be quick or developers won't do them while thorough checks are, by nature, slow.

Our solution is to provide shorthand functions that wrap `devtools::check()` and pass it a variety of different arguments.

Usage

```
check(pkg = ".")

check_fast(pkg = ".")

check_faster(pkg = ".")

check_fastest(pkg = ".")

check_slow(pkg = ".")

check_slower(pkg = ".")

check_slowest(pkg = ".")
```

Arguments

pkg	Package location passed to `devtools::check()`.

Details

The table below describes the `args` passed:

```
check_slowest()  | args = c("--run-donttest", "--run-dontrun", "--use-gct")
 check_slower()  | args = c("--run-donttest", "--run-dontrun")
   check_slow()  | args = c("--run-donttest")
       check_()  | args = c()
   check_fast()  | args = c("--ignore-vignettes")
                 | build_args = c("--no-build-vignettes")
 check_faster()  | args = c("--ignore-vignettes", "--no-examples")
                 | build_args = c("--no-build-vignettes", "--no-examples")
check_fastest()  | args = c("--ignore-vignettes", "--no-examples", "--no-manual", "--no-tests")
                 | build_args = c("--no-build-vignettes", "--no-manual")
```

Value

No return.

See Also

check

`parseDatetime` *Parse datetime strings*

Description

Transforms numeric and string representations of Ymd[HMS] datetimes to `POSIXct` format.

Y, Ym, Ymd, YmdH, YmdHM, and YmdHMS formats are understood, where:

Y four digit year

m month number (1-12, 01-12) or english name month (October, oct.)

d day number of the month (0-31 or 01-31)

H hour number (0-24 or 00-24)

M minute number (0-59 or 00-59)

S second number (0-61 or 00-61)

This allows for mixed inputs. For example, 20181012130900, "2018-10-12-13-09-00", and "2018 Oct. 12 13:09:00" will all be converted to the same `POSIXct` datetime. The incoming datetime vector does not need to have a homogeneous format either – "20181012" and "2018-10-12 13:09" can exist in the same vector without issue. All incoming datetimes will be interpreted in the specified timezone.

If `datetime` is a `POSIXct` it will be returned unmodified, and formats not recognized will be returned as `NA`.

Usage

```
parseDatetime(
  datetime = NULL,
  timezone = NULL,
  expectAll = FALSE,
  isJulian = FALSE,
  quiet = TRUE
)
```

Arguments

datetime	Vector of character or integer datetimes in Ymd[HMS] format (or POSIXct).
timezone	Olson timezone used to interpret dates (required).
expectAll	Logical value determining if the function should fail if any elements fail to parse (default FALSE).
isJulian	Logical value determining whether datetime should be interpreted as a Julian date with day of year as a decimal number.
quiet	Logical value passed on to lubridate::parse_date_time to optionally suppress warning messages.

Value

A vector of POSIXct datetimes.

Mazama Science Conventions

Within Mazama Science package, datetimes not in POSIXct format are often represented as decimal values with no separation (ex: 20181012, 20181012130900), either as numerics or strings.

Implementation

parseDatetime is essentially a wrapper around parse_date_time, handling which formats we want to account for.

See Also

parse_date_time for implementation details.

Examples

```
# All y[md-hms] formats are accepted
parseDatetime(2018, timezone = "America/Los_Angeles")
parseDatetime(201808, timezone = "America/Los_Angeles")
parseDatetime(20180807, timezone = "America/Los_Angeles")
parseDatetime(2018080718, timezone = "America/Los_Angeles")
parseDatetime(201808071812, timezone = "America/Los_Angeles")
parseDatetime(20180807181215, timezone = "America/Los_Angeles")
parseDatetime("2018-08-07 18:12:15", timezone = "America/Los_Angeles")

# Julian days are accepeted
parseDatetime(2018219181215, timezone = "America/Los_Angeles",
              isJulian = TRUE)

# Vector dates are accepted and daylight savings is respected
parseDatetime(
  c("2018-10-24 12:00", "2018-10-31 12:00",
    "2018-11-07 12:00", "2018-11-08 12:00"),
  timezone = "America/New_York"
)
```

```
badInput <- c("20181013", NA, "20181015", "181016", "10172018")

# Return a vector with \code{NA} for dates that could not be parsed
parseDatetime(badInput, timezone = "UTC", expectAll = FALSE)

## Not run:
# Fail if any dates cannot be parsed
parseDatetime(badInput, timezone = "UTC", expectAll = TRUE)

## End(Not run)
```

`setIfNull` *Set a variable to a default value if it is NULL*

Description

This function attempts to set a default value for a given target object. If the object is `NULL`, a default value is returned.

When the target object is not `NULL`, this function will try and coerce it to match the type if the default (given by `typeof`). This is useful in situations where we are looking to parse the input as well, such at looking at elements of an API call string and wanting to set the character numbers as actual numeric types.

Not all coercions are possible, however, and if the function encounters one of these (ex: `setIfNull("foo",5)`) the function will fail.

Usage

```
setIfNull(target, default)
```

Arguments

`target`	Object to test if `NULL` (must be length 1).
`default`	Object to return if `target` is `NULL` (must be length one).

Value

If `target` is not `NULL`, then `target` coerced to the type of `default`. Otherwise, `default` is returned.

Possible Coercions

This function checks the type of the target and default as given by `typeof`. Specifically, it accounts for the types:

- `character`
- `integer`

- double
- complex
- logical
- list

R tries to intelligently coerce types, but some coercions from one type to another won't always be possible. Everything can be turned into a character, but only some character objects can become numeric ("7" can, while "hello" cannot). Some other coercions work, but you will lose information in the process. For example, the *double* 5.7 can be coerced into an *integer*, but the decimal portion will be dropped with no rounding. It is important to realize that while it is possible to move between most types, the results are not always meaningful.

Examples

```
setIfNull(NULL, "foo")
setIfNull(10, 0)
setIfNull("15", 0)


# This function can be useful for adding elements to a list
testList <- list("a" = 1, "b" = "baz", "c" = "4")

testList$a <- setIfNull(testList$a, 0)
testList$b <- setIfNull(testList$c, 0)
testList$d <- setIfNull(testList$d, 6)


# Be careful about unintended results
setIfNull("T", FALSE) # This returns `TRUE`
setIfNull(12.8, 5L)   # This returns the integer 12


## Not run:
# Not all coercions are possible
setIfNull("bar", 5)
setIfNull("5i", 0+0i)
setIfNull("t", FALSE)

## End(Not run)
```

`stopIfNull` *Stop if an object is NULL*

Description

This is a convenience function for testing if an object is `NULL`, and providing a custom error message if it is.

Usage

```
stopIfNull(target, msg = NULL)
```

Arguments

target	Object to test if NULL.
msg	Optional custom message to display when target is NULL.

Value

If target is not NULL, target is returned invisibly.

Examples

```
# Return input invisibly if not NULL
x <- stopIfNull(5, msg = "Custom message")
print(x)

# This can be useful when building pipelines
y <- 1:10
y_mean <-
  y %>%
  stopIfNull() %>%
  mean()

## Not run:
testVar <- NULL
stopIfNull(testVar)
stopIfNull(testVar, msg = "This is NULL")

# Make a failing pipeline
z <- NULL
z_mean <-
  z %>%
  stopIfNull("This has failed.") %>%
  mean()

## End(Not run)
```

---

stopOnError *Error message translator*

Description

When writing R code to be used in production systems that work with user supplied input, it is important to enclose chunks of code inside of a try() block. It is equally important to generate error log messages that can be found and understood during an autopsy when something fails

At Mazama Science we have our own internal standard for how to do error handling in a manner that allows us to quickly navigate to the source of errors in a production system.

The example section contains a snippet showing how we use this function.

Usage

```
stopOnError(result, err_msg = "")
```

Arguments

result	Return from a try() block.
err_msg	Custom error message.

Value

Issues a stop() with an appropriate error message.

Examples

```
## Not run:
logger.setup()

# Arbitrarily deep in the stack we might have:
myFunc <- function(x) {
  a <- log(x)
}

userInput <- 10
result <- try({
  myFunc(x=userInput)
}, silent=TRUE)
stopOnError(result)

userInput <- "ten"
result <- try({
  myFunc(x=userInput)
}, silent=TRUE)
stopOnError(result)

result <- try({
  myFunc(x=userInput)
}, silent=TRUE)
stopOnError(result, "Unable to process user input")

## End(Not run)
```

timeRange *Create a POSIXct time range*

Description

Uses incoming parameters to return a pair of POSIXct times in the proper order. Both start and end times will have lubridate::floor_date() applied to get the nearest unit. This can be modified by specifying ceilingStart = TRUE or ceilingEnd = TRUE in which case lubridate::ceiling_date() will be applied.

The required timezone parameter must be one of those found in OlsonNames.

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

- "YYYYmmddHH[MMSS]"
- "YYYY-mm-dd HH:MM:SS"

Usage

```
timeRange(
  starttime = NULL,
  endtime = NULL,
  timezone = NULL,
  unit = "sec",
  ceilingStart = FALSE,
  ceilingEnd = FALSE
)
```

Arguments

starttime	Desired start datetime (ISO 8601).
endtime	Desired end datetime (ISO 8601).
timezone	Olson timezone used to interpret dates (required).
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 vector of two POSIXcts.

POSIXct inputs

When startdate or enddate are already POSIXct values, they are converted to the timezone specified by timezone without altering the physical instant in time the input represents. This is different from the behavior of parse_date_time (which powers this function), which will force POSIXct inputs into a new timezone, altering the physical moment of time the input represents.

Examples

```
timeRange("2019-01-08 10:12:15", 20190109102030, timezone = "UTC")
```

`timeStamp` *Character representation of a POSIXct*

Description

Uses incoming parameters to return a pair of `POSIXct` times in the proper order. Both start and end times will have `lubridate::floor_date()` applied to get the nearest `unit` unless `ceilingEnd = TRUE` in which case the end time will will have `lubridate::ceiling_date()` applied.

The required `timezone` parameter must be one of those found in `OlsonNames`.

Formatting output is are affected by both `style`:

- `"ymdhms"`
- `"julian"`
- `"clock"`

and `unit` which determines the temporal precision of the generated representation:

- `"year"`
- `"month"`
- `"day"`
- `"hour"`
- `"min"`
- `"sec"`

If ‘style == "julian"‘ && ‘unit = "month"‘, the timestamp will contain the Julian day associated with the beginning of the month.

Usage

```
timeStamp(datetime = NULL, timezone = NULL, unit = "sec", style = "ymdhms")
```

Arguments

`datetime`	Vector of character or integer datetimes in Ymd[HMS] format (or POSIXct).
`timezone`	Olson timezone used to interpret incoming dates (required).
`unit`	Units used to determine precision of generated time stamps.
`style`	Style of representation, Default = "ymdhms".

Value

A vector of time stamps.

POSIXct inputs

When `startdate` or `enddate` are already `POSIXct` values, they are converted to the timezone specified by `timezone` without altering the physical instant in time the input represents. This is different from the behavior of `parse_date_time` (which powers this function), which will force `POSIXct` inputs into a new timezone, altering the physical moment of time the input represents.

Examples

```
datetime <- parseDatetime("2019-01-08 12:30:15", timezone = "UTC")

timeStamp(datetime, "UTC", unit = "year")
timeStamp(datetime, "UTC", unit = "month")
timeStamp(datetime, "UTC", unit = "month", style = "julian")
timeStamp(datetime, "UTC", unit = "day")
timeStamp(datetime, "UTC", unit = "day", style = "julian")
timeStamp(datetime, "UTC", unit = "hour")
timeStamp(datetime, "UTC", unit = "min")
timeStamp(datetime, "UTC", unit = "sec")
timeStamp(datetime, "UTC", unit = "sec", style = "julian")
timeStamp(datetime, "UTC", unit = "sec", style = "clock")
timeStamp(datetime, "America/Los_Angeles", unit = "sec", style = "clock")
```

`timezoneLintRules` *Rules for timezone linting.*

Description

This set of rules is for use with the `lintFunctionArgs_~()`. functions. It includes all time-related functions from the **base** and **lubridate** packages that are involved with parsing or formatting datetimes and helps check whether the appropriate timezone arguments are being explicitly used.

Usage

```
timezoneLintRules
```

Format

A list of function = argument pairs.

Index