Package 'MazamaCoreUtils'

May 7, 2026

Type Package

Version 0.6.2

Title Utility Functions for Production R Code

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, cache management and date-time parsing. Functions for date-time parsing and formatting require that time zones be specified explicitly, avoiding a common source of error when working with environmental time series.

License GPL-3

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

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

Depends R (>= 4.0.0)

Imports devtools, dplyr, geohashTools, logger, lubridate, magrittr, purrr, rlang (>= 1.1.0), rvest, stringr, tibble, xml2

Suggests knitr, markdown, testthat (>= 3.1.7), rmarkdown, roxygen2

Encoding UTF-8

VignetteBuilder knitr

RoxygenNote 7.3.3

NeedsCompilation no

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

Repository CRAN

Date/Publication 2026-05-06 05:10:36 UTC

Contents

`APIKeys` *API keys for data services*

Description

Internal session state used to store API keys for web services.

Format

A named list of character strings.

Details

Users can set API keys with `setAPIKey()`. Keys are remembered for the duration of the R session and can be retrieved with `getAPIKey()`.

This provides a small abstraction layer for dependent packages so that data access functions can test for and retrieve provider-specific API keys with generic code.

See Also

`getAPIKey()`, `setAPIKey()`, `showAPIKeys()`

---

createLocationID *Create one or more unique location IDs*

Description

Create a location ID for each longitude/latitude pair using a geohash.

Usage

```
createLocationID(
  longitude = NULL,
  latitude = NULL,
  precision = 10,
  algorithm = c("geohash", "digest"),
  invalidID = as.character(NA)
)
```

Arguments

`longitude`	Vector of longitudes in decimal degrees east.
`latitude`	Vector of latitudes in decimal degrees north.
`precision`	Precision used when encoding geohashes.
`algorithm`	Encoding algorithm to use. Only `"geohash"` is currently supported. `"digest"` is deprecated and will generate an error.
`invalidID`	Identifier to use for invalid locations. This can be a character string or `NA`.

Details

Each location ID is unique within a geohash grid cell. The `precision` argument determines the size of the grid cell. At the equator, approximate grid cell widths are:

precision	maximum grid cell width
5	~ 4.9 km
6	~ 1.2 km
7	~ 153 m
8	~ 38 m
9	~ 4.8 m
10	~ 1.2 m

Invalid locations are assigned the value specified by `invalidID`, typically `NA`.

Value

Character vector of location IDs.

References

https://michaelchirico.github.io/geohashTools/index.html

Examples

```
longitude <- c(-122.5, 0, NA, -122.5, -122.5)
latitude <- c(47.5, 0, 47.5, NA, 47.5)

createLocationID(longitude, latitude)
createLocationID(longitude, latitude, precision = 7)
createLocationID(longitude, latitude, invalidID = "bad")
```

---

createLocationMask *Create a mask of valid locations*

Description

Create a logical mask identifying valid longitude/latitude pairs.

Usage

```
createLocationMask(
  longitude = NULL,
  latitude = NULL,
  lonRange = c(-180, 180),
  latRange = c(-90, 90),
  removeZeroZero = TRUE
)
```

Arguments

longitude	Vector of longitudes in decimal degrees east.
latitude	Vector of latitudes in decimal degrees north.
lonRange	Range of valid longitudes.
latRange	Range of valid latitudes.
removeZeroZero	Logical specifying whether the coordinate pair (0, 0) should be treated as invalid.

Details

The returned logical vector contains TRUE for valid locations and FALSE for invalid locations. This is useful for filtering data frames to retain only records with valid geographic coordinates.

Longitude and latitude values are considered valid when they:

- fall within lonRange and latRange
- are not missing
- are not located at (0, 0) when removeZeroZero = TRUE

The lonRange and latRange arguments can be used to restrict valid locations to a rectangular geographic region.

Value

Logical vector identifying valid locations.

Examples

```
createLocationMask(
  longitude = c(-120, NA, -120, -220, -120, 0),
  latitude = c(45, 45, NA, 45, 100, 0)
)

createLocationMask(
  longitude = -120:-90,
  latitude = 20:50,
  lonRange = c(-110, -100),
  latRange = c(30, 40)
)
```

dateRange *Create a POSIXct date range*

Description

Create a two-element POSIXct vector representing a date/time range in a specified timezone.

Usage

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

Arguments

startdate	Desired start datetime.
enddate	Desired end datetime.
timezone	Olson timezone used to interpret incoming dates.
unit	Temporal precision used for the returned end-of-range value. One of "day", "hour", "min", or "sec".
ceilingStart	Logical specifying whether to round startdate up to the next day boundary instead of down.
ceilingEnd	Logical specifying whether to include the entirety of the final day.
days	Number of days to include when either startdate or enddate is omitted.

Details

The returned range is ordered from earliest to latest. The first element represents the beginning of the requested date range and the second element represents the end of the requested date range at the requested temporal precision.

By default, the returned end time is one unit *before* the beginning of enddate. For example:

```
dateRange(20190101, 20190102, timezone = "UTC")
[1] "2019-01-01 00:00:00 UTC"
[2] "2019-01-01 23:59:59 UTC"
```

Setting ceilingEnd = TRUE includes the entirety of enddate:

```
dateRange(
  20190101,
  20190101,
  timezone = "UTC",
  ceilingEnd = TRUE
)
[1] "2019-01-01 00:00:00 UTC"
[2] "2019-01-01 23:59:59 UTC"
```

The `ceilingEnd` argument addresses ambiguity in phrases such as `"August 1-8"`. With `ceilingEnd = FALSE` (default), the range extends through the end of August 7, stopping at the midnight boundary where August 8 begins. With `ceilingEnd = TRUE`, the range extends through the end of August 8.

Input dates are parsed with `parseDatetime()` using the specified `timezone`.

Value

Two-element `POSIXct` vector ordered from earliest to latest.

Default arguments

If either `startdate` or `enddate` is missing, the missing boundary is calculated using `days`.

If both are missing, `enddate` defaults to the current day in `timezone` and `startdate` is calculated as `enddate - days`.

End-of-day units

The returned end time is adjusted to the last representable value within the requested unit:

`unit = "day"` End time is midnight at the start of the final day.

`unit = "hour"` End time is `23:00:00`.

`unit = "min"` End time is `23:59:00`.

`unit = "sec"` End time is `23:59:59`.

POSIXct inputs

When `startdate` or `enddate` are already `POSIXct` values, they are first converted to `timezone` with `lubridate::with_tz()` without changing the represented instant in time.

Parameter precedence

When parameters conflict, the following rules apply:

1. If both `startdate` and `enddate` are supplied, `days` is ignored.
2. If `startdate` is missing, `ceilingStart` is ignored.
3. If `enddate` is missing, `ceilingEnd` is ignored.

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

Create a sequence of local-midnight `POSIXct` datetimes in a specified timezone.

Usage

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

Arguments

`startdate`	Desired start datetime.
`enddate`	Desired end datetime.
`timezone`	Olson timezone used to interpret incoming dates.
`ceilingEnd`	Logical specifying whether to include the end of the final day.

Details

The returned sequence begins at midnight local time on `startdate` and ends at midnight local time on `enddate`, *i.e.* the *beginning* of `enddate`.

The ceilingEnd argument addresses ambiguity in date ranges such as "August 1-8". With ceilingEnd = FALSE (default), the sequence ends at the beginning of August 8. With ceilingEnd = TRUE, the sequence includes the entirety of August 8, ending at the midnight that begins August 9.

Input dates are parsed with parseDatetime() using the specified timezone. Any hour-minute-second information is removed after parsing.

Value

A vector of POSIXct datetimes at local midnight.

POSIXct inputs

When startdate or enddate are already POSIXct values, they are first converted to timezone with lubridate::with_tz() without changing the represented instant in time. They are then floored to local midnight.

Note

This function preserves local clock-time midnight boundaries across daylight savings transitions. This differs from seq.Date(..., by = "day"), which advances by fixed 24-hour intervals and can drift away from midnight local time during daylight savings changes.

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

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

# POSIXct inputs preserve the represented instant before flooring
jst <- dateSequence(
  20190307,
  20190315,
```

```
  timezone = "Asia/Tokyo"
)

jst

dateSequence(
  jst[1],
  jst[7],
  timezone = "UTC"
)
```

`getAPIKey` *Get API key*

Description

Return the API key associated with a web service provider.

Usage

```
getAPIKey(provider = NULL)
```

Arguments

`provider`	Web service provider.

Details

If `provider = NULL`, all currently stored API keys are returned.

Value

API key string, `NULL`, or a named list of all provider/key pairs.

See Also

APIKeys, `setAPIKey()`, `showAPIKeys()`

`html_getLinks` *Extract links from an HTML page*

Description

Parse an HTML page and return all `<a href="...">...</a>` links as a data frame.

Usage

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

html_getLinkNames(url = NULL)

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

Arguments

`url`	URL or local file path of an HTML page.
`relative`	Logical specifying whether to return relative URLs. If `FALSE`, relative URLs are converted to absolute URLs using `url` as the base.

Details

The returned data frame contains the human-readable link text in `linkName` and the `href` value in `linkUrl`. This is useful for extracting links from index pages, including web-accessible directories that list downloadable files.

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

Value

A tibble with `linkName` and `linkUrl` columns.

`html_getLinkNames()` returns a character vector of link names.

`html_getLinkUrls()` returns a character vector of link URLs.

Examples

```
## Not run:

# If you want to download lots of USCensus shapefiles
url <- "https://www2.census.gov/geo/tiger/GENZ2019/shp/"

browseURL(url)

dataLinks <- html_getLinks(url)

dataLinks <-
```

```
  dataLinks %>%
  dplyr::filter(stringr::str_detect(linkName, "us_county"))

head(dataLinks, 10)

html_getLinkNames(url)
html_getLinkUrls(url, relative = FALSE)

## End(Not run)
```

`html_getTables` *Extract tables from an HTML page*

Description

Parse an HTML page and return all <table> elements as a list of data frames.

Usage

```
html_getTables(url = NULL, header = NA)

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

Arguments

`url`	URL or local file path of an HTML page.
`header`	Logical specifying whether the first row should be used as column names. If `NA`, the first row is used only when it contains <th> elements.
`index`	Index identifying which table to return.

Details

The `url` argument may be either a remote URL or a local file path. Tables are parsed with `rvest::html_table()`. To extract a single table, use `html_getTable()`.

Value

List of data frames, one for each HTML table.

A single data frame containing the requested HTML table.

Examples

```
## Not run:
url <- "https://en.wikipedia.org/wiki/List_of_tz_database_time_zones"

tables <- html_getTables(url)
firstTable <- tables[[1]]

head(firstTable)
nrow(firstTable)

## End(Not run)
```

`initializeLogging` *Initialize standard log files*

Description

Create a standard set of MazamaCoreUtils log files.

Usage

```
initializeLogging(logDir = NULL, filePrefix = "", createDir = TRUE)
```

Arguments

`logDir`	Directory in which to write log files.
`filePrefix`	Character string prepended to log file names.
`createDir`	Logical specifying whether to create `logDir` if it does not already exist.

Details

This convenience function creates or validates a log directory, archives any existing standard log files by appending a UTC timestamp, and then initializes logging with `logger.setup()`.

Standard log files include:

```
TRACE.log
DEBUG.log
INFO.log
WARN.log
ERROR.log
```

When `filePrefix` is supplied, it is prepended to each log file name.

Value

No return value. Called for side effects.

See Also

logger.setup()

`lintFunctionArgs` *Lint function calls for required named arguments*

Description

Parse R source code and identify calls to selected functions that are missing required named arguments.

Usage

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

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

Arguments

`filePath`	Path to a single R source file.
`rules`	Named list of linting rules. Each list name is a function name and each value is a character vector of required named arguments.
`fullPath`	Logical specifying whether returned file paths should be absolute paths. If `FALSE`, only base file names are returned.
`dirPath`	Path to a directory containing R source files.

Details

Rules are supplied as a named list where each name is a function to check and each value is a character vector of required argument names. A function call passes when all required arguments are supplied by name.

This linter only checks whether arguments are named in the call. It does not evaluate code, inspect argument values, or detect unnamed positional arguments.

Value

A tibble describing matching function calls, with columns:

file Source file path or file name.

line_number Line number where the function call begins.

column_number Column number where the function call begins.

function_name Name of the function being checked.

named_args List column containing named arguments used in the call.

includes_required Logical indicating whether all required named arguments were supplied.

Limitations

This linter only detects named arguments. For example, `foo(x = bar, "baz")` is treated as specifying the named argument `x`, but the value `bar` and the unnamed argument `"baz"` are not inspected.

Examples

```
## Not run:
rules <- list(
  fn_one = "x",
  fn_two = c("foo", "bar")
)

lintFunctionArgs_file(
  filePath = "local_test/timezone_lint_test_script.R",
  rules = rules
)

lintFunctionArgs_dir(
  dirPath = "./R",
  rules = MazamaCoreUtils::timezoneLintRules
)

## End(Not run)
```

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

Description

Load a pre-generated R binary data file from either a local directory or a remote URL.

Usage

```
loadDataFile(
  filename = NULL,
  dataUrl = NULL,
  dataDir = NULL,
  priority = c("dataDir", "dataUrl")
)
```

Arguments

`filename`	Name of the `.rda` file to load.
`dataUrl`	Remote URL directory containing data files.
`dataDir`	Local directory containing data files.
`priority`	First data source to try when both `dataDir` and `dataUrl` are supplied.

Details

This function is intended for use by package-level `*_load()` helper functions. It allows locally cached data files to be used when available, avoiding unnecessary internet access.

If both `dataDir` and `dataUrl` are provided, `priority` determines which source is tried first. If loading from the first source fails, the other source is used as a fallback.

Value

Object loaded from the `.rda` file.

Examples

```
## Not run:
filename <- "USCensusStates_02.rda"
dataDir <- "~/Data/Spatial"
dataUrl <- "http://data.mazamascience.com/MazamaSpatialUtils/Spatial_0.8"

# Load local file
USCensusStates <- loadDataFile(filename, dataDir = dataDir)

# Load remote file
USCensusStates <- loadDataFile(filename, dataUrl = dataUrl)

# Load local file with remote file as backup
USCensusStates <- loadDataFile(
  filename,
  dataDir = dataDir,
  dataUrl = dataUrl,
  priority = "dataDir"
)

# Load remote file with local file as backup
USCensusStates <- loadDataFile(
  filename,
  dataDir = dataDir,
  dataUrl = dataUrl,
  priority = "dataUrl"
)

## End(Not run)
```

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

Description

Emit a `DEBUG` level log message.

Usage

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

Arguments

msg	Message with optional format strings.
...	Additional arguments passed to sprintf() formatting.

Details

Logging must first be initialized with logger.setup().

Value

No return value. Called for side effects.

See Also

logger.setup()

---

logger.error *Python-style logging statements*

Description

Emit an ERROR level log message.

Usage

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

Arguments

msg	Message with optional format strings.
...	Additional arguments passed to sprintf() formatting.

Details

Logging must first be initialized with logger.setup().

Value

No return value. Called for side effects.

See Also

logger.setup()

logger.fatal *Python-style logging statements*

Description

Emit a FATAL level log message.

Usage

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

Arguments

msg	Message with optional format strings.
...	Additional arguments passed to sprintf() formatting.

Details

Logging must first be initialized with logger.setup().

Value

No return value. Called for side effects.

See Also

logger.setup()

logger.info *Python-style logging statements*

Description

Emit an INFO level log message.

Usage

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

Arguments

msg	Message with optional format strings.
...	Additional arguments passed to sprintf() formatting.

Details

Logging must first be initialized with `logger.setup()`.

Value

No return value. Called for side effects.

See Also

`logger.setup()`

`logger.isInitialized` *Check whether logging has been initialized*

Description

Determine whether `logger.setup()` has already been called.

Usage

```
logger.isInitialized()
```

Details

This function is useful in package code that conditionally emits log statements only when logging has been configured.

Value

Logical scalar indicating whether logging has been initialized.

See Also

`logger.setup()`

Examples

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

logger.setup()

logger.isInitialized()

## End(Not run)
```

`logger.setLevel` *Set console log level*

Description

Set the minimum log level displayed in the console.

Usage

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

Arguments

`level`	Logging threshold level.

Details

By default, only `FATAL` messages are displayed in the console. This function allows users to display additional log messages interactively.

Available log levels are:

```
TRACE
DEBUG
INFO
WARN
ERROR
FATAL
```

Value

No return value. Called for side effects.

Note

All functionality is implemented with the excellent **logger** package.

See Also

`logger.setup()`

Examples

```
## Not run:
# Enable console logging
logger.setup()

# Show DEBUG and higher messages in the console
logger.setLevel(DEBUG)
```

```
## End(Not run)
```

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

Description

Configure level-specific log files using the package logging API.

Usage

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

Arguments

`traceLog`	File path receiving `TRACE` messages.
`debugLog`	File path receiving `DEBUG` messages.
`infoLog`	File path receiving `INFO` messages.
`warnLog`	File path receiving `WARN` messages.
`errorLog`	File path receiving `ERROR` messages.
`fatalLog`	File path receiving `FATAL` messages.

Details

Logging is built on top of the logger package while retaining the historical MazamaCoreUtils logging interface.

Separate log files can be created for different log levels so that, for example, an `errorLog` contains only `ERROR` and `FATAL` messages while a `debugLog` contains `DEBUG` messages as well as all higher-severity messages.

Any log file argument left as `NULL` is disabled and no file will be created for that level.

After initialization, logging statements can be generated with: `logger.trace()`, `logger.debug()`, `logger.info()`, `logger.warn()`, `logger.error()`, and `logger.fatal()`.

Log messages are formatted with:

```
LEVEL [YYYY-MM-DD HH:MM:SS UTC] message
```

Console logging is enabled by default only for `FATAL` messages. Use `logger.setLevel()` to display additional log messages in the console.

Value

No return value. Called for side effects.

Note

All functionality is implemented with the excellent **logger** package.

See Also

logger.trace(), logger.debug(), logger.info(), logger.warn(), logger.error(), logger.fatal()

Examples

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

# Generate log messages
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 risky")

result <- try(1 / "a", silent = TRUE)
logger.error("error message: %s", geterrmessage())
logger.fatal("fatal statement: %s", "THE END")

cat(readLines("debug.log"), sep = "\n")
cat(readLines("info.log"), sep = "\n")
cat(readLines("error.log"), sep = "\n")

## End(Not run)
```

---

logger.trace *Python-style logging statements*

Description

Emit a TRACE level log message.

Usage

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

Arguments

msg	Message with optional format strings.
...	Additional arguments passed to sprintf() formatting.

Details

Logging must first be initialized with logger.setup().

Value

No return value. Called for side effects.

See Also

logger.setup()

---

logger.warn *Python-style logging statements*

Description

Emit a WARN level log message.

Usage

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

Arguments

msg	Message with optional format strings.
...	Additional arguments passed to sprintf() formatting.

Details

Logging must first be initialized with logger.setup().

Value

No return value. Called for side effects.

See Also

logger.setup()

logLevels *Log levels*

Description

Logging level constants used by the MazamaCoreUtils logging API.

Usage

```
FATAL
```

Format

An object of class `integer` of length 1.

Details

Available log levels include:

```
FATAL
ERROR
WARN
INFO
DEBUG
TRACE
```

These constants are retained for backwards compatibility with the original MazamaCoreUtils logging system.

manageCache *Manage cache size*

Description

Remove old or excess files from a cache directory.

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 used to order files for size-based removal. One of "atime", "ctime", or "mtime".
maxFileAge	Maximum file age in days. Files with modification times older than this value are removed regardless of cache size. Fractional days are allowed.

Details

Files are eligible for removal when their extension matches extensions. Matching is case-sensitive and extensions may be supplied with or without a leading dot.

Files can be removed for two reasons:

- files older than maxFileAge days are removed first
- if the remaining cache exceeds maxCacheSize, additional files are removed until the cache is under the requested size

When removing files to satisfy maxCacheSize, files are ordered by the timestamp specified by sortBy.

Timestamp meanings are:

atime File access time, updated when a file is opened.

ctime File change time, updated when file metadata changes.

mtime File modification time, updated when file contents change.

Value

Invisibly returns the number of files removed.

Examples

```
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, pattern = "\\.csv$", full.names = TRUE)) {
  print(file.info(file)[, c("size", "mtime")])
}

# Remove files based on access time until the cache is under 1 MB
manageCache(
  CACHE_DIR,
  extensions = "csv",
```

```
  maxCacheSize = 1,
  sortBy = "atime"
)

for (file in list.files(CACHE_DIR, pattern = "\\.csv$", full.names = TRUE)) {
  print(file.info(file)[, c("size", "mtime")])
}
```

---

packageCheck *Run package checks*

Description

Convenience wrappers around `devtools::check()` for package checking at different levels of thoroughness.

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

These functions make it easy to run quick checks during active development and more thorough checks before merging or releasing package changes.

The functions are ordered from most thorough to fastest:

`check_slowest()` Builds the manual, runs `donttest` and `dontrun` examples, and uses `--use-gct`.

`check_slower()` Builds the manual and runs `donttest` and `dontrun` examples.

`check_slow()` Builds the manual and runs `donttest` examples.

`check()` Standard development check without building the manual or running `donttest` examples.

`check_fast()` Skips vignette building and ignores vignettes during checking.

`check_faster()` Skips vignette building, ignores vignettes, and skips examples.

`check_fastest()` Skips vignette building, ignores vignettes, skips examples, and skips tests.

Value

Invisibly returns the result from `devtools::check()`.

See Also

`devtools::check()`

`parseDatetime` *Parse datetimes*

Description

Convert character, numeric, integer, or `POSIXct` datetimes to `POSIXct`.

Usage

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

Arguments

`datetime`	Vector of character, numeric, integer, or `POSIXct` datetimes.
`timezone`	Olson timezone used to interpret incoming datetimes.
`expectAll`	Logical value specifying whether to stop if any non-missing input values fail to parse.
`isJulian`	Logical value specifying whether `datetime` should be interpreted as a Julian date using day-of-year notation.
`quiet`	Logical value passed to `lubridate::parse_date_time()` to suppress parsing warnings.

Details

This function accepts a variety of compact date/time formats commonly used in Mazama Science packages, including `Y`, `Ym`, `Ymd`, `YmdH`, `YmdHM`, and `YmdHMS`. Inputs may be mixed within the same vector.

Examples of equivalent inputs include:

```
20181012130900
"2018-10-12-13-09-00"
"2018 Oct. 12 13:09:00"
```

All incoming datetimes are interpreted in the specified `timezone`. If `datetime` is already `POSIXct`, it is converted to the requested timezone with `lubridate::with_tz()`.

If a character datetime includes signed offset information, such as `"-07:00"`, that offset is used by `lubridate::parse_date_time()` when determining the equivalent instant.

Value

A `POSIXct` vector.

Mazama Science conventions

Within Mazama Science packages, datetimes not already in `POSIXct` format are often represented as compact decimal values with no separators, such as `20181012` or `20181012130900`, either as numbers or strings.

Implementation

`parseDatetime()` is a wrapper around `lubridate::parse_date_time()` that defines the datetime formats supported by MazamaCoreUtils.

See Also

`lubridate::parse_date_time()`

Examples

```
# All Y[mdHMS] 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")
parseDatetime("2018-08-07 18:12:15-07:00", timezone = "UTC")

# Julian days are accepted
parseDatetime(
```

```
  2018219181215,
  timezone = "America/Los_Angeles",
  isJulian = TRUE
)

# Mixed vector inputs are accepted
parseDatetime(
  c("2018-10-24 12:00", "201810311200", "2018-11-07 12:00"),
  timezone = "America/New_York"
)

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

# Return NA for dates that cannot be parsed
parseDatetime(badInput, timezone = "UTC", expectAll = FALSE)

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

## End(Not run)
```

`setAPIKey` *Set API key*

Description

Set the API key associated with a web service provider.

Usage

```
setAPIKey(provider = NULL, key = NULL)
```

Arguments

`provider`	Web service provider.
`key`	API key.

Details

API keys are stored in package session state and are remembered only for the duration of the current R session.

Value

Invisibly returns the previous value of the API key.

See Also

getAPIKey(), showAPIKeys()

---

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

Description

Returns default when target is NULL; otherwise returns target unchanged.

Usage

```
setIfNull(target, default, enforcedType = NULL)
```

Arguments

target	Object to test for NULL.
default	Object to return when target is NULL.
enforcedType	Optional character string specifying the suffix of an as.*() coercion function to apply to the returned value. For example, "double" uses as.double(), "character" uses as.character(), and "Date" uses as.Date(). If NULL (the default), no coercion is performed.

Details

This is useful for assigning default values to optional arguments while preserving any user-supplied value exactly as provided.

Optionally, enforcedType may be used to coerce the returned value to a specific type. This coercion is applied after the NULL check and affects both target and default.

Value

The value of target if it is not NULL; otherwise default.

If enforcedType is specified, the returned value is coerced using the corresponding as.*() function.

Examples

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

# User-supplied values are returned unchanged
setIfNull("15", 0)
setIfNull("mean", 0)
setIfNull(mean, 0)
```

```
# Optional type enforcement
setIfNull("15", 0, enforcedType = "double")
setIfNull(NULL, "15", enforcedType = "integer")
```

`showAPIKeys` *Show API keys*

Description

Print all currently set API keys.

Usage

```
showAPIKeys()
```

Value

No return value. Called for side effects.

See Also

`getAPIKey()`, `setAPIKey()`

`stopIfNull` *Stop if an object is* `NULL`

Description

Convenience function for validating that an object is not `NULL`.

Usage

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

Arguments

`target`	Object to test.
`msg`	Optional error message to display if `target` is `NULL`. Must be a character string of length one.

Details

If `target` is not `NULL`, it is returned invisibly. If `target` is `NULL`, the function stops with either a default or user-supplied error message.

This function is especially useful for validating required function arguments or for guarding intermediate results in pipelines.

Value

Invisibly returns `target` when it is not `NULL`.

Examples

```
# Return input invisibly if not NULL
x <- stopIfNull(5)
print(x)

# Useful in pipelines
y <- 1:10
y_mean <-
  y %>%
  stopIfNull() %>%
  mean()

## Not run:
# Trigger the default error message
testVar <- NULL
stopIfNull(testVar)

# Trigger a custom error message
stopIfNull(testVar, msg = "This is NULL")

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

## End(Not run)
```

---

stopOnError *Stop on try-error*

Description

Generate a consistent error message from the result of a `try()` block.

Usage

```
stopOnError(
  result,
  err_msg = "",
  prefix = "",
  maxLength = 500,
  truncatedLength = 120,
```

```
  call. = FALSE
)
```

Arguments

result	Return value from a try() block.
err_msg	Optional custom error message.
prefix	Optional text to prepend to the error message.
maxLength	Maximum allowed error message length before truncation.
truncatedLength	Length of the truncated error message.
call.	Logical indicating whether the call should be included in the error message. Passed to stop().

Details

This function is intended for production code where potentially fragile operations are wrapped in try(..., silent = TRUE). If result inherits from "try-error", a cleaned and optionally customized error message is generated and passed to stop().

If result is not a "try-error", the function returns NULL.

Value

Returns NULL if result is not a "try-error"; otherwise stops with an error.

Note

If logging has been initialized, the final error message is logged with logger.error() before calling stop().

Examples

```
## Not run:
myFunc <- function(x) {
  log(x)
}

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

stopOnError(result)

try({
  myFunc("ten")
}, silent = TRUE) %>%
  stopOnError(err_msg = "Unable to process user input")

try({
```

```
  myFunc("ten")
}, silent = TRUE) %>%
  stopOnError(
    prefix = "USER_INPUT_ERROR",
    maxLength = 40,
    truncatedLength = 32
  )

## End(Not run)
```

---

timeRange *Create a POSIXct time range*

Description

Create an ordered two-element POSIXct time range from start and end datetime values.

Usage

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

Arguments

starttime	Desired start datetime.
endtime	Desired end datetime.
timezone	Olson timezone used to interpret incoming datetimes.
unit	Unit used for rounding. Passed to lubridate::floor_date() or lubridate::ceiling_date().
ceilingStart	Logical specifying whether to round the start time up instead of down.
ceilingEnd	Logical specifying whether to round the end time up instead of down.

Details

Input values are converted with parseDatetime() using the required timezone argument. The resulting start and end times are sorted so the earlier time is always returned first.

By default, both times are rounded down with lubridate::floor_date() using the requested unit. Set ceilingStart = TRUE or ceilingEnd = TRUE to round either endpoint up with lubridate::ceiling_date() instead.

Value

Two-element POSIXct vector ordered from earliest to latest.

POSIXct inputs

When startdate or enddate are already POSIXct values, they are first converted to timezone with lubridate::with_tz() without changing the represented instant in time.

Examples

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

timeRange(
  starttime = "2019-01-08 10:12:15",
  endtime = "2019-01-09 10:20:30",
  timezone = "UTC",
  unit = "hour"
)
```

---

timeStamp *Create character timestamps*

Description

Convert datetimes to compact character timestamps suitable for file names, identifiers, labels, and other reproducible text output.

Usage

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

Arguments

datetime	Vector of character, integer, or POSIXct datetimes.
timezone	Olson timezone used to interpret incoming datetimes.
unit	Temporal precision of the generated timestamp.
style	Output timestamp style.

Details

Input values are converted with `parseDatetime()` using the required `timezone` argument. When `datetime = NULL`, the current UTC time is used and `timezone` defaults to `"UTC"`.

The `unit` argument controls the precision of the output timestamp. The `style` argument controls the output format.

Supported `unit` values are:

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

Supported `style` values are:

```
"ymdhms"   compact calendar time
"ymdThms"  compact calendar time with "T" separator
"julian"   year and Julian day
"clock"    ISO-like clock time
```

For `style = "julian"` and `unit = "month"`, the timestamp uses the Julian day associated with the beginning of the month.

Value

Character vector of timestamps.

POSIXct inputs

When `startdate` or `enddate` are already `POSIXct` values, they are first converted to `timezone` with `lubridate::with_tz()` without changing the represented instant in time.

Examples

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

timeStamp()
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 = "ymdThms")
timeStamp(datetime, "UTC", unit = "sec", style = "julian")
```

```
timeStamp(datetime, "UTC", unit = "sec", style = "clock")
timeStamp(datetime, "America/Los_Angeles", unit = "sec", style = "clock")
timeStamp(datetime, "America/Los_Angeles", unit = "msec", style = "clock")
```

`timezoneLintRules` *Timezone linting rules*

Description

Rules used by `lintFunctionArgs_file()` and `lintFunctionArgs_dir()` to find date/time function calls that should explicitly specify timezone arguments.

Usage

```
timezoneLintRules
```

Format

A named list of function/argument pairs.

Details

Each list name is a function to check. Each value is the required named timezone-related argument for that function.

Entries with `"DEPRECATED"` are used to flag functions that should generally be avoided in package code because they depend on the local system clock or timezone.

Examples

```
str(timezoneLintRules)
```

`validateLonLat` *Validate longitude and latitude values*

Description

Validate a single longitude/latitude pair to ensure both values are numeric scalars and fall within valid geographic bounds.

Usage

```
validateLonLat(longitude = NULL, latitude = NULL)
```

Arguments

longitude	Single longitude in decimal degrees east.
latitude	Single latitude in decimal degrees north.

Details

Longitudes must fall between -180 and 180 degrees and latitudes must fall between -90 and 90 degrees. If validation fails, an error is generated.

Value

Invisibly returns TRUE if validation succeeds.

Examples

```
validateLonLat(-122.5, 47.5)

## Not run:
validateLonLat(-200, 47.5)
validateLonLat(-122.5, NA)

## End(Not run)
```

---

validateLonsLats *Validate longitude and latitude vectors*

Description

Validate longitude and latitude vectors to ensure they are numeric, have matching lengths, and contain values within valid geographic bounds.

Usage

```
validateLonsLats(longitude = NULL, latitude = NULL, na.rm = FALSE)
```

Arguments

longitude	Vector of longitudes in decimal degrees east.
latitude	Vector of latitudes in decimal degrees north.
na.rm	Logical specifying whether to remove NA values before validation.

Details

Longitudes must fall between -180 and 180 degrees and latitudes must fall between -90 and 90 degrees. If validation fails, an error is generated.

Value

Invisibly returns TRUE if validation succeeds.

Examples

```
longitude <- c(-122.5, -122.4)
latitude <- c(47.5, 47.6)

validateLonsLats(longitude, latitude)

# Remove missing values before validation
validateLonsLats(
  c(-122.5, NA),
  c(47.5, NA),
  na.rm = TRUE
)

## Not run:
validateLonsLats(c(-200, 0), c(45, 46))

## End(Not run)
```

Index