Package 'MazamaCoreUtils'

December 3, 2018

Type Package

Version 0.1.3

Title Utility Functions for Production R Code

Author Jonathan Callahan [aut, cre],
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), futile.logger

Imports dplyr, lubridate, stringr

Suggests knitr, markdown, testthat, rmarkdown

Encoding UTF-8

VignetteBuilder knitr

LazyData true

RoxygenNote 6.1.0

NeedsCompilation no

Repository CRAN

Date/Publication 2018-12-03 05:10:18 UTC

R topics documented:

`initializeLogging` *Initialize standard log files*

Description

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

```
result <- try({
  Copy and old log files
  timestamp <- strftime(lubridate::now(), "
  for ( logLevel in c("TRACE","DEBUG","INFO","ERROR") ) {
    oldFile <- file.path(logDir,paste0(logLevel,".log"))
    newFile <- file.path(logDir,paste0(logLevel,".log.",timestamp))
    if ( file.exists(oldFile) ) {
      file.rename(oldFile, newFile)
    }
  }
}, silent=TRUE)
stopOnError(result, "Could not rename old log files.")

result <- try({
  # Set up logging
  logger.setup(traceLog = file.path(logDir, "TRACE.log"),
               debugLog=file.path(logDir, "DEBUG.log"),
               infoLog=file.path(logDir, "INFO.log"),
               errorLog=file.path(logDir, "ERROR.log"))
}, silent=TRUE)
stopOnError(result, "Could not create log files.")
```

Usage

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

Arguments

`logDir`	Directory in which to write log files.

Value

No return value.

`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.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, 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)])
}
```

`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)
```

Index