Package 'this.path'

December 2, 2023

Version 2.3.0

Date 2023-12-02

License MIT + file LICENSE

Title Get Executing Script's Path

Description Determine the path of the executing script. Compatible with a few popular GUIs: 'Rgui', 'RStudio', 'VSCode', 'Jupyter', 'Emacs', and 'Rscript' (shell). Compatible with several functions and packages: 'source()', 'sys.source()', 'debugSource()' in 'RStudio', 'compiler::loadcmp()', 'box::use()', 'knitr::knit()', 'plumber::plumb()', 'shiny::runApp()', 'package:targets', and 'testthat::source_file()'.

Author Iris Simmons

Maintainer Iris Simmons <ikwsimmo@gmail.com>

Suggests utils, microbenchmark

Enhances compiler, box, IRkernel, jsonlite, knitr, plumber, rprojroot, rstudioapi, shiny, targets, testthat

URL https://github.com/ArcadeAntics/this.path

BugReports https://github.com/ArcadeAntics/this.path/issues

ByteCompile TRUE

Biarch TRUE

Type Package

R topics documented:

`this.path-package` *Get Script's Path*

Description

Determine the path of the executing script. Compatible with a few popular GUIs: 'Rgui', 'RStudio', 'VSCode', 'Jupyter', 'Emacs', and '`Rscript`' (shell). Compatible with several functions and packages: `source()`, `sys.source()`, `debugSource()` in 'RStudio', `compiler::loadcmp()`, `box::use()`, `knitr::knit()`, `plumber::plumb()`, `shiny::runApp()`, **targets**, and `testthat::source_file()`.

Details

The most important functions from **this.path** are `this.path()`, `this.dir()`, `here()`, and `this.proj()`:

- `this.path()` returns the normalized path of the script in which it is written.
- `this.dir()` returns the directory of `this.path()`.
- `here()` constructs file paths against `this.dir()`.
- `this.proj()` constructs file paths against the project root of `this.dir()`.

this.path also provides functions for constructing and manipulating file paths:

- `path.join()`, `basename2()`, and `dirname2()` are drop in replacements for `file.path()`, `basename()`, and `dirname()` which better handle drives and network shares.
- `splitext()`, `removeext()`, `ext()`, and `ext<-()` split a path into root and extension, remove a file extension, get an extension, or set an extension for a file path.
- `path.split()`, `path.split.1()`, and `path.unsplit()` split the path to a file into components.
- `relpath()`, `rel2here()`, and `rel2proj()` turn absolute paths into relative paths.

New additions to **this.path** include:

- `with_site.file()` and `with_init.file()` allow `this.path()` and related to be used in the site-wide startup profile file or a user profile.
- `LINENO()` returns the line number of the executing expression.
- `set.sys.path()` implements `this.path()` for any `source()`-like functions outside of `source()`, `sys.source()`, `debugSource()` in 'RStudio', `compiler::loadcmp()`, `box::use()`, `knitr::knit()`, `plumber::plumb()`, `shiny::runApp()`, **targets**, and `testthat::source_file()`.
- `shFILE()` looks through the command line arguments, extracting 'FILE' from either of the following: '`-f`' '`FILE`' or '`--file=FILE`'

Note

This package started from a stack overflow posting, found at:

https://stackoverflow.com/questions/1815606/determine-path-of-the-executing-script

If you like this package, please consider upvoting my answer so that more people will see it! If you have an issue with this package, please use `utils::bug.report(package = "this.path")` to report your issue.

Author(s)

Iris Simmons

Maintainer: Iris Simmons <ikwsimmo@gmail.com>

`basename2` *Manipulate File Paths*

Description

`basename2()` removes all of the path up to and including the last path separator (if any).

`dirname2()` returns the part of the `path` up to but excluding the last path separator, or `"."` if there is no path separator.

Usage

```
basename2(path)
dirname2(path)
```

Arguments

`path`	character vector, containing path names.

Details

Tilde-expansion (see `?path.expand()`) of the path will be performed.

Trailing path separators are removed before dissecting the path, and for `dirname2()` any trailing file separators are removed from the result.

Value

A character vector of the same length as `path`.

Behaviour on Windows

If path is an empty string, then both dirname2() and basename2() return an emty string.

\ and / are accepted as path separators, and dirname2() does **NOT** translate the path separators.

Recall that a network share looks like "//host/share" and a drive looks like "d:".

For a path which starts with a network share or drive, the path specification is the portion of the string immediately afterward, e.g. "/path/to/file" is the path specification of "//host/share/path/to/file" and "d:/path/to/file". For a path which does not start with a network share or drive, the path specification is the entire string.

The path specification of a network share will always be empty or absolute, but the path specification of a drive does not have to be, e.g. "d:file" is a valid path despite the fact that the path specification does not start with "/".

If the path specification of path is empty or is "/", then dirname2() will return path and basename2() will return an empty string.

Behaviour under Unix-alikes

If path is an empty string, then both dirname2() and basename2() return an emty string.

Recall that a network share looks like "//host/share".

For a path which starts with a network share, the path specification is the portion of the string immediately afterward, e.g. "/path/to/file" is the path specification of "//host/share/path/to/file". For a path which does not start with a network share, the path specification is the entire string.

If the path specification of path is empty or is "/", then dirname2() will return path and basename2() will return an empty string.

Examples

```
path <- c("/usr/lib", "/usr/", "usr", "/", ".", "..")
x <- cbind(path, dirname = dirname2(path), basename = basename2(path))
print(x, quote = FALSE, print.gap = 3)
```

---

check.path *Check 'this.path()' is Functioning Correctly*

Description

Add check.path("path/to/file") to the start of your script to initialize this.path() and check that it is returning the expected path.

Usage

```
check.path(...)
check.dir(...)

check.proj(...)
```

Arguments

...	further arguments passed to path.join() which must return a character string; the path you expect this.path() or this.dir() to return. The specified path can be as deep as necessary (just the basename, the last directory and the basename, the last two directories and the basename, ...), but do not use an absolute path. this.path() makes R scripts portable, but using an absolute path in check.path() or check.dir() makes an R script non-portable, defeating a major purpose of this package.

Details

check.proj() is a specialized version of check.path() that checks the path all the way up to the project's directory.

Value

If the expected path / / directory matches this.path() / / this.dir(), then TRUE invisibly.

Otherwise, an error is thrown.

Examples

```
# ## I have a project called 'EOAdjusted'
# ##
# ## Within this project, I have a folder called 'code'
# ## where I place all of my scripts.
# ##
# ## One of these scripts is called 'provrun.R'
# ##
# ## So, at the top of that R script, I could write:
#
#
# this.path::check.path("EOAdjusted", "code", "provrun.R")
#
# ## or:
#
# this.path::check.path("EOAdjusted/code/provrun.R")
```

---

ext *File Extensions*

Description

splitext() splits an extension from a path.

removeext() removes an extension from a path.

ext() gets the extension of a path.

ext<-() sets the extension of a path.

Usage

```
splitext(path, compression = FALSE)
removeext(path, compression = FALSE)
ext(path, compression = FALSE)
ext(path, compression = FALSE) <- value
```

Arguments

path	character vector, containing path names.
compression	should compression extensions ".gz", ".bz2", and ".xz" be taken into account when removing / / getting an extension?
value	a character vector, typically of length 1 or length(path), or NULL.

Details

Tilde-expansion (see ?path.expand()) of the path will be performed.

Trailing path separators are removed before dissecting the path.

Except for path <- NA_character_, it will always be true that path == paste0(removeext(path), ext(path)).

Value

for splitext(), a matrix with 2 rows and length(path) columns. The first row will be the roots of the paths, the second row will be the extensions of the paths.

for removeext() and ext(), a character vector the same length as path.

for ext<-(), the updated object.

Examples

```
splitext(character(0))
splitext("")

splitext("file.ext")

path <- c("file.tar.gz", "file.tar.bz2", "file.tar.xz")
splitext(path, compression = FALSE)
splitext(path, compression = TRUE)

path <- "this.path_1.0.0.tar.gz"
ext(path) <- ".png"
path

path <- "this.path_1.0.0.tar.gz"
ext(path, compression = TRUE) <- ".png"
path
```

FILE *Macros in Package 'this.path'*

Description

FILE() and LINE() are intended to be used in a similar manner to the macros __FILE__ and __LINE__ in C. They are useful for generating a diagnostic message / / warning / / error to about the status of the program.

Usage

```
FILE()
LINE()
```

Examples

```
FILE.R <- tempfile(fileext = ".R")
writeLines("fun <- function ()
{
    message(sprintf('invalid value %d at %s, line %d',
        -1, FILE(), LINE()))
}
", FILE.R)
source(FILE.R, verbose = FALSE, keep.source = TRUE)
fun()
unlink(FILE.R)
```

---

from.shell *Top-Level Code Environment*

Description

Determine if a program is the main program, or if an R script was run from a shell.

Usage

```
from.shell()
is.main()
```

Details

When an R script is run from a shell, `from.shell()` and `is.main()` will both be `TRUE`. If that script sources another R script, `from.shell()` and `is.main()` will both be `FALSE` for the duration of the second script.

Otherwise, `from.shell()` will be `FALSE`. `is.main()` will be `TRUE` when there is no executing script or when `source()`-ing a script in a toplevel context, and `FALSE` otherwise.

Value

`TRUE` or `FALSE`.

Examples

```
FILES <- tempfile(c("file1_", "file2_"), fileext = ".R")
this.path:::.write.code({
    from.shell()
    is.main()
}, FILES[2])
this.path:::.write.code((
    bquote(this.path:::.withAutoprint({
        from.shell()
        is.main()
        source(.(FILES[2]), echo = TRUE, verbose = FALSE,
            prompt.echo = "file2> ", continue.echo = "file2+ ")
    }, spaced = TRUE, verbose = FALSE, width.cutoff = 60L,
       prompt.echo = "file1> ", continue.echo = "file1+ "))
), FILES[1])
```

```
this.path:::.Rscript(
    c("--default-packages=this.path", "--vanilla", FILES[1])
)


this.path:::.Rscript(c("--default-packages=this.path", "--vanilla",
    "-e", "cat(\"\n> from.shell()\\n\")",
    "-e", "from.shell()",
    "-e", "cat(\"\n> is.main()\\n\")",
    "-e", "is.main()",
    "-e", "cat(\"\n> source(commandArgs(TRUE)[[1L]])\\n\")",
    "-e", "source(commandArgs(TRUE)[[1L]])",
    FILES[1]))


unlink(FILES)
```

---

getinitwd *Get Initial Working Directory*

Description

`getinitwd()` returns an absolute filepath representing the working directory at the time of loading this package.

Usage

```
getinitwd()
initwd
```

Value

`getinitwd()` returns a character string or `NULL` if the initial working directory is not available.

Examples

```
cat("\ninitial working directory:\n"); getinitwd()
cat("\ncurrent working directory:\n"); getwd()
```

---

here *Construct Path to File, Starting With Script's Directory*

Description

`here()` constructs file paths starting with `this.dir()`.

`this.proj()` constructs file paths starting with the project root of `this.dir()`.

`reset.proj()` resets the path cache of `this.proj()`. This can be useful if you create a new project that you would like to be detected without restarting your R session.

Usage

```
here(..., local = FALSE, n = 0,
     envir = parent.frame(n + 1),
     matchThisEnv = getOption("topLevelEnvironment"),
     srcfile = if (n) sys.parent(n) else 0, .. = 0)

this.proj(..., local = FALSE, n = 0,
          envir = parent.frame(n + 1),
          matchThisEnv = getOption("topLevelEnvironment"),
          srcfile = if (n) sys.parent(n) else 0)

reset.proj()

## alias for 'here'
ici(..., local = FALSE, n = 0,
    envir = parent.frame(n + 1),
    matchThisEnv = getOption("topLevelEnvironment"),
    srcfile = if (n) sys.parent(n) else 0, .. = 0)
```

Arguments

`...`	further arguments passed to `path.join()`.
`local, n, envir, matchThisEnv, srcfile`	See ?`this.path()`.
`..`	the number of directories to go back.

Details

For `this.proj()`, the project root has the same criterion as `here::here()`, but unlike `here::here()`, `this.proj()` supports sub-projects and multiple projects in use at once. Additionally, `this.proj()` is independent of working directory, whereas `here::here()` relies on the working directory being set somewhere within the project when `package:here` is loaded. Arguably, this makes it better than `here::here()`.

Value

A character vector of the arguments concatenated term-by-term.

Examples

```
tmpdir <- tempfile(pattern = "dir")
dir.create(tmpdir)

writeLines("this file signifies that its directory is the project root",
    this.path::path.join(tmpdir, ".here"))

FILE.R <- this.path::path.join(tmpdir, "src", "R", "script1.R")
dir.create(this.path::dirname2(FILE.R), recursive = TRUE)
this.path:::.write.code({
    this.path::this.path()
    if (requireNamespace("rprojroot"))
        this.path::this.proj()
    ## use 'here' to run another script located nearby
```

```
    this.path::here("script2.R")
    ## or maybe to read input from a file
    this.path::here(.. = 2, "input", "data1.csv")
    ## but sometimes it is easier to use the project root
    ## this allows you to move the R script up or down
    ## a directory without changing the .. number
    if (requireNamespace("rprojroot"))
        this.path::this.proj("input", "data1.csv")
}, FILE.R)

source(FILE.R, echo = TRUE)

unlink(tmpdir, recursive = TRUE)
```

LINENO *Line Number of Executing Expression*

Description

Get the line number of the executing expression.

Usage

```
LINENO(n = 0, envir = parent.frame(n + 1),
       matchThisEnv = getOption("topLevelEnvironment"),
       srcfile = if (n) sys.parent(n) else 0)
```

Arguments

`n`, `envir`, `matchThisEnv`, `srcfile`
: See ?`this.path()`.

Details

`LINENO()` starts by examining argument `srcfile`. If it is a `srcref` or contains one, it will return the `first_line` number.

If it does not find a line number associated with `srcfile`, it will look for a path in a similar manner to `this.path()` except it skips the `srcfile` and GUI aspects. If it finds a path, it will look for the most recent expression that has a source reference and has a source file equal to the path. If it finds one, it will return the `first_line` number.

If it does not find a line number, it will return `NA_integer_`.

Value

integer.

Note

LINENO() only works if the expressions have a srcref.

Scripts run with Rscript do not store their srcref, even when getOption("keep.source") is TRUE.

For source() and sys.source(), make sure to supply argument keep.source = TRUE directly, or set options "keep.source" and "keep.source.pkgs" to TRUE.

For debugSource() in 'RStudio', it has no argument keep.source, so set option "keep.source" to TRUE before calling.

For compiler::loadcmp(), the srcref is never stored for the compiled code, there is nothing that can be done.

For knitr::knit(), the srcref is never stored, there is nothing that can be done. I am looking into a fix.

For **targets**, set option "keep.source" to TRUE before calling.

For box::use(), plumber::plumb(), shiny::runApp(), and testthat::source_file(), the srcref is always stored.

Examples

```
FILE.R <- tempfile(fileext = ".R")
writeLines(c("
LINENO()
LINENO()
## LINENO() respects #line directives
#line 15
LINENO()
#line 1218
cat(sprintf('invalid value %d at %s, line %d\\n',
            -5, try.this.path(), LINENO()))
"), FILE.R)


# ## previously used:
#
# source(FILE.R, echo = TRUE, verbose = FALSE,
#     max.deparse.length = Inf, keep.source = TRUE)
#
# ## but it echoes incorrectly with #line directives
this.path:::.source(FILE.R, echo = TRUE, verbose = FALSE,
    max.deparse.length = Inf, keep.source = TRUE)


unlink(FILE.R)
```

OS.type *Detect the Operating System Type*

Description

OS.type is a list of TRUE / / FALSE values dependent on the platform under which this package was built.

Usage

```
OS.type
```

Value

A list with at least the following components:

`AIX`	Built under IBM AIX.
`HPUX`	Built under Hewlett-Packard HP-UX.
`linux`	Built under some distribution of Linux.
`darwin`	Built under Apple OSX and iOS (Darwin).
`iOS.simulator`	Built under iOS in Xcode simulator.
`iOS`	Built under iOS on iPhone, iPad, etc.
`macOS`	Built under OSX.
`solaris`	Built under Solaris (SunOS).
`cygwin`	Built under Cygwin POSIX under Microsoft Windows.
`windows`	Built under Microsoft Windows.
`win64`	Built under Microsoft Windows (64-bit).
`win32`	Built under Microsoft Windows (32-bit).
`UNIX`	Built under a UNIX-style OS.

Source

http://web.archive.org/web/20191012035921/http://nadeausoftware.com/articles/2012/01/c_c_tip_how_use_compiler_p

---

`path.functions` *Constructs Path Functions Similar to 'this.path()'*

Description

`path.functions()` accepts a pathname and constructs a set of path-related functions, similar to `this.path()` and associated.

Usage

```
path.functions(file, local = FALSE, n = 0,
               envir = parent.frame(n + 1),
               matchThisEnv = getOption("topLevelEnvironment"),
               srcfile = if (n) sys.parent(n) else 0)
```

Arguments

`file`	a character string giving the pathname of the file or URL.
`local, n, envir, matchThisEnv, srcfile`	See `?this.path()`.

Value

An environment with at least the following bindings:

`this.path`	Function with formals `(original = FALSE, contents = FALSE)` which returns the normalized file path, the original file path, or the contents of the file.
`this.dir`	Function of no arguments which returns the directory of the normalized file path.
`here`	Function with formals `(..., .. = 0)` which constructs file paths, starting with the file's directory.
`this.proj`	Function with formals `(..., .. = 0)` which constructs file paths, starting with the project root.
`rel2here, rel2proj`	Functions with formals `(path)` which turn absolute paths into relative paths, against the file's directory / / project root.
`LINENO`	Function with formals `NULL` which returns the line number of the executing expression in `file`.

`path.join` *Construct Path to File*

Description

Construct the path to a file from components / / paths in a platform-**DEPENDENT** way.

Usage

```
path.join(...)
```

Arguments

`...`	character vectors.

Details

When constructing a path to a file, the last absolute path is selected and all trailing components are appended. This is different from `file.path()` where all trailing paths are treated as components.

Value

A character vector of the arguments concatenated term-by-term and separated by `"/"`.

Examples

```
path.join("C:", "test1")

path.join("C:/", "test1")

path.join("C:/path/to/file1", "/path/to/file2")

path.join("//host-name/share-name/path/to/file1", "/path/to/file2")

path.join("C:testing", "C:/testing", "~", "~/testing", "//host",
```

```
    "//host/share", "//host/share/path/to/file", "not-an-abs-path")

path.join("c:/test1", "c:test2", "C:test3")

path.join("test1", "c:/test2", "test3", "//host/share/test4", "test5",
    "c:/test6", "test7", "c:test8", "test9")
```

`path.split` *Split File Path Into Individual Components*

Description

Split the path to a file into components in a platform-**DEPENDENT** way.

Usage

```
path.split(path)
path.split.1(path)
path.unsplit(...)
```

Arguments

`path`	character vector.
`...`	character vectors, or one list of character vectors.

Value

for `path.split()`, a list of character vectors.

for `path.split.1()` and `path.unsplit()`, a character vector.

Note

`path.unsplit()` is **NOT** the same as `path.join()`.

Examples

```
path <- c(
    NA,
    "",
    paste0("https://raw.githubusercontent.com/ArcadeAntics/PACKAGES/",
           "src/contrib/Archive/this.path/this.path_1.0.0.tar.gz"),
    "\\\\host\\share\\path\\to\\file",
    "\\\\host\\share\\",
    "\\\\host\\share",
    "C:\\path\\to\\file",
    "C:path\\to\\file",
    "path\\to\\file",
    "\\path\\to\\file",
    "~\\path\\to\\file",
    ## paths with character encodings
    `Encoding<-`("path/to/fil\xe9", "latin1"),
    "C:/Users/iris/Documents/\u03b4.R"
)
print(x <- path.split(path))
print(path.unsplit(x))
```

pragma_once *Evaluate a File Once*

Description

pragma_once() is intended to be used in a similar manner to the preprocessor directive #pragma once in C.

Usage

```
pragma_once(expr)
```

Arguments

expr

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (expr)
{
    path <- .External2(.C_this.path)
    if (.pragma_once(path)) {
        rm(path)
        expr
    }
  }
```

print.ThisPathDocumentContext *Printing "ThisPathDocumentContext" Objects*

Description

Print a "ThisPathDocumentContext" object.

Usage

```
## S3 method for class 'ThisPathDocumentContext'
print(x, ..., quote = TRUE)

## S3 method for class 'ThisPathDocumentContext'
format(x, ...)

## S3 method for class 'ThisPathDocumentContext'
as.character(x, ...)
```

Arguments

x	object of class "ThisPathDocumentContext".
...	unused.
quote	logical, indicating whether or not strings should be printed with surrounding quotes.

Details

An object of class "ThisPathDocumentContext" is generated upon calling set.sys.path(), wrap.source(), sys.path(), env.path(), or src.path(), and by extension this.path(). It contains information about the path of the executing / / current script.

These objects are not usually user-visible.

Value

for print.ThisPathDocumentContext(), x invisibly.

for format.ThisPathDocumentContext(), a character vector of lines.

for as.character.ThisPathDocumentContext(), a character string of concatenated lines.

Examples

```
fun <- function (file)
{
    set.sys.path(file, Function = "fun")
    `.this.path::document.context`
}


fun()


fun("clipboard")


fun(paste0("https://raw.githubusercontent.com/ArcadeAntics/",
           "this.path/main/tests/sys-path-with-urls.R"))


FILE.R <- tempfile(fileext = ".R"); file.create(FILE.R)

x <- fun(FILE.R)
print(x)
print(x, quote = FALSE)
format(x)
as.character(x)

unlink(FILE.R)
```

progArgs *Providing Arguments to a Script*

Description

`withArgs()` allows you to `source()` an R script while providing arguments. As opposed to running with `Rscript`, the code will be evaluated in the same session in an environment of your choosing.

`fileArgs()` // `progArgs()` are generalized versions of `commandArgs(trailingOnly = TRUE)`, allowing you to access the script's arguments whether it was sourced or run from a shell.

`asArgs()` coerces R objects into a character vector, for use with command line applications and `withArgs()`.

Usage

```
asArgs(...)
fileArgs()
progArgs()
withArgs(...)
```

Arguments

`...`	R objects to turn into script arguments; typically `logical`, `numeric`, `character`, `Date`, and `POSIXt` vectors. for `withArgs()`, the first argument should be an (unevaluated) call to `source()`, `sys.source()`, `debugSource()` in 'RStudio', `compiler::loadcmp()`, `knitr::knit()`, `testthat::source_file()`, or a `source()`-like function containing `set.sys.path()` // `wrap.source()`.

Details

`fileArgs()` will return the arguments associated with the executing script, or `character(0)` when there is no executing script.

`progArgs()` will return the arguments associated with the executing script, or `commandArgs(trailingOnly = TRUE)` when there is no executing script.

`asArgs()` coerces objects into command-line arguments. `...` is first put into a list, and then each non-list element is converted to character. They are converted as follows:

Factors (class `"factor"`**)** using `as.character.factor()`

Date-Times (class `"POSIXct"` **and** `"POSIXlt"`**)** using format `"%Y-%m-%d %H:%M:%OS6"` (retains as much precision as possible)

Numbers (class `"numeric"` **and** `"complex"`**)** with 17 significant digits (retains as much precision as possible) and `"."` as the decimal point character.

Raw Bytes (class `"raw"`**)** using `sprintf("0x%02x", )` (for easy conversion back to raw with `as.raw()` or `as.vector(, "raw")`)

All others will be converted to character using `as.character()` and its methods.

The arguments will then be unlisted, and all attributes will be removed. Arguments that are `NA_character_` after conversion will be converted to `"NA"` (since the command-line arguments also never have missing strings).

Value

for `asArgs()`, `fileArgs()`, and `progArgs()`, a character vector.

for `withArgs()`, the result of evaluating the first argument.

Examples

```
this.path::asArgs(NULL, c(TRUE, FALSE, NA), 1:5, pi, exp(6i),
    letters[1:5], as.raw(0:4), Sys.Date(), Sys.time(),
    list(list(list("lists are recursed"))))


FILE.R <- tempfile(fileext = ".R")
this.path:::.write.code({
    this.path:::.withAutoprint({
        this.path::sys.path()
        this.path::fileArgs()
        this.path::progArgs()
    }, spaced = TRUE, verbose = FALSE, width.cutoff = 60L)
}, FILE.R)


## wrap your source call with a call to withArgs()
this.path::withArgs(
    source(FILE.R, local = TRUE, verbose = FALSE),
    letters[6:10], pi, exp(1)
)
this.path::withArgs(
    sys.source(FILE.R, environment()),
    letters[11:15], pi + 1i * exp(1)
)
this.path:::.Rscript(c("--default-packages=NULL", "--vanilla", FILE.R,
    this.path::asArgs(letters[16:20], pi, Sys.time())))
## fileArgs() will be character(0) because there is no executing script
this.path:::.Rscript(c("--default-packages=NULL", "--vanilla",
    rbind("-e", readLines(FILE.R)[-2L]),
    this.path::asArgs(letters[16:20], pi, Sys.time())))


# ## with R >= 4.1.0, use the forward pipe operator '|>' to
# ## make calls to withArgs() more intuitive:
# source(FILE.R, local = TRUE, verbose = FALSE) |> this.path::withArgs(
#     letters[6:10], pi, exp(1))
# sys.source(FILE.R, environment()) |> this.path::withArgs(
#     letters[11:15], pi + 1i * exp(1))


## withArgs() also works with set.sys.path() and wrap.source()
sourcelike <- function (file, envir = parent.frame())
{
    file <- set.sys.path(file)
    envir <- as.environment(envir)
    exprs <- parse(n = -1, file = file)
    for (i in seq_along(exprs)) eval(exprs[i], envir)
}
this.path::withArgs(sourcelike(FILE.R), letters[21:26])
```

```
sourcelike2 <- function (file, envir = parent.frame())
{
    envir <- as.environment(envir)
    exprs <- parse(n = -1, file = file)
    for (i in seq_along(exprs)) eval(exprs[i], envir)
}
sourcelike3 <- function (file, envir = parent.frame())
{
    envir <- as.environment(envir)
    wrap.source(sourcelike2(file = file, envir = envir))
}
this.path::withArgs(sourcelike3(FILE.R), LETTERS[1:5])
this.path::withArgs(wrap.source(sourcelike2(FILE.R)), LETTERS[6:10])


unlink(FILE.R)
```

`relpath` *Make a Path Relative to Another*

Description

When working with **this.path**, you will be dealing with a lot of absolute paths. These paths are not portable for saving within files nor tables, so convert them to relative paths with `relpath()`.

Usage

```
relpath(path, relative.to = normalizePath(getwd(), "/", TRUE))

rel2here(path, local = FALSE, n = 0, envir = parent.frame(n + 1),
         matchThisEnv = getOption("topLevelEnvironment"),
         srcfile = if (n) sys.parent(n) else 0)

rel2proj(path, local = FALSE, n = 0,
         envir = parent.frame(n + 1),
         matchThisEnv = getOption("topLevelEnvironment"),
         srcfile = if (n) sys.parent(n) else 0)
```

Arguments

`path`	character vector of file / / URL pathnames.
`relative.to`	character string; the file / / URL pathname to make `path` relative to.
`local, n, envir, matchThisEnv, srcfile`	See ?`this.path()`.

Details

Tilde-expansion (see ?`path.expand()`) is first done on `path` and `relative.to`.

If `path` and `relative.to` are equivalent, `"."` will be returned. If `path` and `relative.to` have no base in common, the normalized `path` will be returned.

Value

character vector of the same length as path.

Examples

```
## Not run:
relpath(
    c(
        ## paths which are equivalent will return "."
        "C:/Users/effective_user/Documents/this.path/man",


        ## paths which have no base in common return as themselves
        paste0("https://raw.githubusercontent.com/ArcadeAntics/",
               "this.path/main/tests/sys-path-with-urls.R"),
        "D:/",
        "//host-name/share-name/path/to/file",


        "C:/Users/effective_user/Documents/testing",
        "C:\\Users\\effective_user",
        "C:/Users/effective_user/Documents/R/thispath.R"
    ),
    relative.to = "C:/Users/effective_user/Documents/this.path/man"
)

## End(Not run)
```

---

set.gui.path *Declare GUI's Active Document*

Description

set.gui.path() can be used to implement sys.path() for arbitrary GUIs. This is similar to set.sys.path() which can be used to implement sys.path() for arbitrary source()-like functions.

Usage

```
set.gui.path(...)

thisPathNotExistsError(..., call. = TRUE, domain = NULL,
                       call = .getCurrentCall())
```

Arguments

..., call., domain, call	See details.

Details

`thisPathNotExistsError()` is provided for use inside `set.gui.path()`, and should not be used elsewhere.

If no arguments are passed to `set.gui.path()`, the default behaviour will be restored.

If one argument is passed to `set.gui.path()`, it must be a function that returns the path of the active document in your GUI. It must accept the following arguments: `(verbose, original, for.msg, contents)` (default values are unnecessary). This makes sense for a GUI which can edit and run R code from several different documents such as RGui, RStudio, VSCode, and Emacs.

If two or three arguments are passed to `set.gui.path()`, they must be the name of the GUI, the path of the active document, and optionally a function to read the contents of the document. If provided, the function must accept at least one argument which will be the normalized path of the document. This makes sense for a GUI which can edit and run R code from only one document such as Jupyter and shell.

It is best to call this function as a user hook.

```
setHook(packageEvent("this.path"),
function(pkgname, pkgpath)
{
    this.path::set.gui.path(<...>)
}, action = "prepend")
```

An example for a GUI which can run code from multiple documents:

```
evalq(envir = new.env(parent = .BaseNamespaceEnv), {
    .guiname <- "myGui"
    .custom.gui.path <- function(verbose, original, for.msg, contents) {
        if (verbose)
            cat("Source: document in", .guiname, "\n")

        ## your GUI needs to know which document is active
        ## and some way to retrieve that document from R
        doc <- <.myGui.activeDocument()>

        ## if no documents are open, 'doc' should be NULL
        ## or some other object to represent no documents open
        if (is.null(doc)) {
            if (for.msg)
                NA_character_
            else stop(this.path::thisPathNotExistsError("R is running from ",
                .guiname, " with no documents open\n",
                " (or document has no path)"))
        }
        else if (contents) {
            ## somehow, get and return the contents of the document
            <doc$contents>
        }
        else {
            ## somehow, get the path of the document
            path <- <doc$path>
            if (nzchar(path)) {
                ## if the path is not normalized, this will normalize it
```

```
                if (isTRUE(original))
                    path
                else normalizePath(path, "/", TRUE)
                # ## otherwise, you could just do:
                # path
            }
            else if (for.msg)
                ## return "Untitled" possibly translated
                gettext("Untitled", domain = "RGui", trim = FALSE)
            else stop("document in ", .guiname, " does not exist")
        }
    }
    ## recommended to prevent tampering
    lockEnvironment(environment(), bindings = TRUE)
    setHook(packageEvent("this.path"),
    function(pkgname, pkgpath) {
        this.path::set.gui.path(.custom.gui.path)
    }, action = "prepend")
})
```

An example for a GUI which can run code from only one document:

```
evalq(envir = new.env(parent = .BaseNamespaceEnv), {
    .guiname <- "myGui"
    .path <- "~/example.R"
    .custom.readContents <- function(path) {
        ## get the contents of the document
        readLines(path, warn = FALSE)
    }
    ## recommended to prevent tampering
    lockEnvironment(environment(), bindings = TRUE)
    setHook(packageEvent("this.path"), function(pkgname, pkgpath) {
        this.path::set.gui.path(.guiname, .path, .custom.readContents)
    }, action = "prepend")
    # ## if your GUI does not have/need a .custom.readContents
    # ## function, then this works just as well:
    # setHook(packageEvent("this.path"), function(pkgname, pkgpath) {
    #     this.path::set.gui.path(.guiname, .path)
    # }, action = "prepend")
})
```

`set.jupyter.path` *Declare Executing 'Jupyter' Notebook's Filename*

Description

`sys.path()` does some guess work to determine the path of the executing notebook in 'Jupyter'. This involves listing all the files in the initial working directory, filtering those which are R notebooks, then filtering those with contents matching the top-level expression.

This could possibly select the wrong file if the same top-level expression is found in another file. As such, you can use `set.jupyter.path()` to declare the executing 'Jupyter' notebook's filename.

Usage

```
set.jupyter.path(...)
set.sys.path.jupyter(...)
```

Arguments

...	further arguments passed to path.join(). If no arguments are provided or exactly one argument is provided that is NA or NULL, the 'Jupyter' path is unset.

Details

This function may only be called from a top-level context in 'Jupyter'. It is recommended that you do **NOT** provide an absolute path. Instead, provide just the basename and the directory will be determined by the initial working directory.

Value

character string, invisibly; the declared path for 'Jupyter'.

Examples

```
# ## if you opened the file "~/file50b816a24ec1.ipynb", the initial
# ## working directory should be "~". You can write:
#
# set.jupyter.path("file50b816a24ec1.ipynb")
#
# ## and then sys.path() will return "~/file50b816a24ec1.ipynb"
```

---

shFILE *Get 'FILE' Provided to R by a Shell*

Description

Look through the command line arguments, extracting 'FILE' from either of the following: '-f' 'FILE' or '--file=FILE'

Usage

```
shFILE(original = FALSE, for.msg = FALSE, default, else.)

site.file(original = FALSE, for.msg = FALSE, default, else.)
init.file(original = FALSE, for.msg = FALSE, default, else.)
```

Arguments

original	TRUE, FALSE, or NA; should the original or the normalized path be returned? NA means the normalized path will be returned if it has already been forced, and the original path otherwise.
for.msg	TRUE or FALSE; do you want the path for the purpose of printing a diagnostic message / / warning / / error? for.msg = TRUE will ignore original = FALSE, and will use original = NA instead.

`default`	if 'FILE' is not found, this value is returned.
`else.`	missing or a function to apply if 'FILE' is found. See `tryCatch2()` for inspiration.

Value

character string, or `default` if 'FILE' was not found.

Note

The original and the normalized path are saved; this makes them faster when called subsequent times.

On Windows, the normalized path will use `/` as the file separator.

See Also

`this.path()`, `here()`

Examples

```
FILE.R <- tempfile(fileext = ".R")
this.path:::.write.code({
    this.path:::.withAutoprint({
        shFILE(original = TRUE)
        shFILE()
        shFILE(default = {
            stop("since 'FILE.R' will be found, argument 'default'\n",
                 " will not be evaluated, so this error will not be\n",
                 " thrown! you can use this to your advantage in a\n",
                 " similar manner, doing arbitrary things only if\n",
                 " 'FILE.R' is not found")
        })
    }, spaced = TRUE, verbose = FALSE, width.cutoff = 60L)
}, FILE.R)
this.path:::.Rscript(
    c("--default-packages=this.path", "--vanilla", FILE.R)
)
unlink(FILE.R)


for (expr in c("shFILE(original = TRUE)",
               "shFILE(original = TRUE, default = NULL)",
               "shFILE()",
               "shFILE(default = NULL)"))
{
    cat("\n\n")
    this.path:::.Rscript(
        c("--default-packages=this.path", "--vanilla", "-e", expr)
    )
}
```

source.exprs *Evaluate and Print Expressions*

Description

source.exprs() evaluates and auto-prints expressions as if in a toplevel context.

Usage

```
source.exprs(exprs, evaluated = FALSE, envir = parent.frame(),
             echo = TRUE, print.eval = TRUE)
```

Arguments

exprs, evaluated, envir, echo, print.eval
: See withAutoprint().

Sys.putenv *Set Environment Variables*

Description

Sys.putenv() sets environment variables (for other processes called from within R or future calls to Sys.getenv() from this R process).

Usage

```
Sys.putenv(x)
```

Arguments

x
: a character vector, or an object coercible to character. Strings must be of the form "name=value".

Value

A logical vector, with elements being true if setting the corresponding variable succeeded.

See Also

Sys.setenv()

Examples

```
Sys.putenv(c("R_TEST=testit", "A+C=123"))
Sys.getenv("R_TEST")
Sys.unsetenv("R_TEST")  ## under Unix-alikes may warn and not succeed
Sys.getenv("R_TEST", unset = NA)
```

`sys.srcref` *Get 'srcref' From Call Stack*

Description

This is the R-level version of the function that implements `this.path()`. It retrieves the `srcref` of the call `n` generations back from `sys.srcref()`.

Usage

```
sys.srcref(n = 1, which = if (n) sys.parent(n) else 0)
```

Arguments

`n`	See `?this.path()`. However, note that `n` defaults to `0` in `this.path()` but defaults to `1` here.
`which`	the frame number to inspect for source references. An alternative to specifying `n`.

Value

A `srcref` object or `NULL`.

Examples

```
## this example will not work with 'Run examples'
## which uses 'package:knitr' since knitted
## documents do not store source references
fun <- function ()
{
    list(
        `sys.srcref()` = this.path::sys.srcref(),
        ## while this might seem like a simpler alternative,
        ## you will see it does not work in a couple cases below
        `attr(sys.call(sys.parent()), "srcref")` =
            attr(sys.call(sys.parent()), "srcref")
    )
}


## the braces are unnecessary when using example("sys.srcref"),
## but are needed when copied into the R Console
{ fun() }
{ print(fun()) }
{ try(print(fun())) }
```

this.path *Determine Script's Filename*

Description

`this.path()` returns the normalized path of the script in which it was written.

`this.dir()` returns the directory of `this.path()`.

Usage

```
this.path(verbose = getOption("verbose"), original = FALSE,
          for.msg = FALSE, contents = FALSE, local = FALSE,
          n = 0, envir = parent.frame(n + 1),
          matchThisEnv = getOption("topLevelEnvironment"),
          srcfile = if (n) sys.parent(n) else 0,
          default, else.)

this.dir(verbose = getOption("verbose"), local = FALSE,
         n = 0, envir = parent.frame(n + 1),
         matchThisEnv = getOption("topLevelEnvironment"),
         srcfile = if (n) sys.parent(n) else 0,
         default, else.)
```

Arguments

`verbose`	`TRUE` or `FALSE`; should the method in which the path was determined be printed?
`original`	`TRUE`, `FALSE`, or `NA`; should the original or the normalized path be returned? `NA` means the normalized path will be returned if it has already been forced, and the original path otherwise.
`for.msg`	`TRUE` or `FALSE`; do you want the path for the purpose of printing a diagnostic message / / warning / / error? This will return `NA_character_` in most cases where an error would have been thrown. `for.msg = TRUE` will ignore `original = FALSE`, and will use `original = NA` instead.
`contents`	`TRUE` or `FALSE`; should the contents of the script be returned instead? In 'Jupyter', a list of character vectors will be returned, the contents separated into cells. Otherwise, if `for.msg` is `TRUE` and the executing script cannot be determined, `NULL` will be returned. Otherwise, a character vector will be returned. You could use `as.character(unlist(this.path(contents = TRUE)))` if you require a character vector. This is intended for logging purposes. This is useful in 'Rgui', 'RStudio', 'VSCode', and 'Emacs' when the source document has contents but no path.
`local`	`TRUE` or `FALSE`; should the search for the executing script be confined to the local environment in which `set.sys.path()` was called?
`n`	the number of additional generations to go back. By default, `this.path()` will look for a path based on the `srcref` of the call to `this.path()` and the environment in which `this.path()` was called. This can be changed to be based on the `srcref` of the call and the calling environment `n` generations up the stack. See section **Argument 'n'** for more details.

`envir, matchThisEnv`
: arguments passed to `topenv()` to determine the top level environment in which to search for an associated path.

`srcfile`
: source file in which to search for a pathname, or an object containing a source file. This includes a source reference, a call, an expression object, or a closure.

`default`
: this value is returned if there is no executing script.

`else.`
: function to apply if there is an executing script. See `tryCatch2()` for inspiration.

Details

`this.path()` starts by examining argument `srcfile`. It looks at the bindings `filename` and `wd` to determine the associated file path. Filenames such as `""`, `"clipboard"`, and `"stdin"` will be ignored since they do not refer to files. A source file of class `"srcfilecopy"` in which binding `isFile` is `FALSE` will also be ignored. A source file of class `"srcfilealias"` will use the aliased `filename` in determining the associated path.

If it does not find a path associated with `srcfile`, it will next examine arguments `envir` and `matchThisEnv`. Specifically, it calculates `topenv(envir, matchThisEnv)` then looks for an associated path. It will find a path associated with the top level environment in two ways:

1. from a **box** module's namespace.
2. from an attribute `"path"`.

If it does not find an associated path with `envir` and `matchThisEnv`, it will next examine the call stack looking for a source call: a call to function `source()`, `sys.source()`, `debugSource()` in 'RStudio', `compiler::loadcmp()`, `box::use()`, `knitr::knit()`, `plumber::plumb()`, `shiny::runApp()`, **targets**, or `testthat::source_file()`. If a source call is found, the file argument is returned from the function's evaluation environment. If you have your own `source()`-like function that you would like to be recognized by `this.path()`, please use `set.sys.path()` / / `wrap.source()` or contact the package maintainer so that it can be implemented.

If no source call is found up the calling stack, it will next examine the GUI in use. If R is running from:

a shell, such as the Windows command-line / / Unix terminal then the shell arguments are searched for '`-f`' '`FILE`' or '`--file=FILE`' (the two methods of taking input from '`FILE`') ('`-f`' '`-`' and '`--file=-`' are ignored). The last '`FILE`' is extracted and returned. If no arguments of either type are supplied, an error is thrown.

If R is running from a shell under a Unix-alike with '`-g`' '`Tk`' or '`--gui=Tk`', an error is thrown. 'Tk' does not make use of its '`-f`' '`FILE`', '`--file=FILE`' arguments.

'Rgui' then the source document's filename (the document most recently interacted with) is returned (at the time of evaluation). Please note that minimized documents *WILL* be included when looking for the most recently used document. It is important to not leave the current document (either by closing the document or interacting with another document) while any calls to `this.path()` have yet to be evaluated in the run selection. If no documents are open or the source document does not exist (not saved anywhere), an error is thrown.

'RStudio' then the active document's filename (the document in which the cursor is active) is returned (at the time of evaluation). If the active document is the R console, the source document's filename (the document open in the current tab) is returned (at the time of evaluation). Please note that the source document will *NEVER* be a document open in another window (with the **Show in new window** button). Please also note that an active document open in another window can sometimes lose focus and become inactive, thus returning the incorrect

path. It is best **NOT** to not run R code from a document open in another window. It is important to not leave the current tab (either by closing or switching tabs) while any calls to `this.path()` have yet to be evaluated in the run selection. If no documents are open or the source document does not exist (not saved anywhere), an error is thrown.

'VSCode' then the source document's filename is returned (at the time of evaluation). It is important to not leave the current tab (either by closing or switching tabs) while any calls to `this.path()` have yet to be evaluated in the run selection. If no documents are open or the source document does not exist (not saved anywhere), an error is thrown.

'Jupyter' then the source document's filename is guessed by looking for R notebooks in the initial working directory, then searching the contents of those files for an expression matching the top-level expression. Please be sure to save your notebook before using `this.path()`, or explicitly use `set.jupyter.path()`.

'Emacs' then the source document's filename is returned (at the time of evaluation). 'Emacs' must be running as a server, either by running `(server-start)` (consider adding to your '`~/.emacs`' file) or typing `M-x server-start`. It is important to not leave the current window (either by closing or switching buffers) while any calls to `this.path()` have yet to be evaluated in the run selection. If multiple frames are active, `this.path()` will pick the first frame containing the corresponding R session.

If multiple 'Emacs' sessions are active, `this.path()` will only work in the primary session due to limitations in 'emacsclient.exe'. If you want to run multiple R sessions, it is better to run one 'Emacs' session with multiple frames, one R session per frame. Use `M-x make-frame` to make a new frame, or `C-x 5 f` to visit a file in a new frame.

Additionally, never use `C-c C-b` to send the current buffer to the R process. This copies the buffer contents to a new file which is then `source()`-ed. The source references now point to the wrong file. Instead, use `C-x h` to select the entire buffer then `C-c C-r` to evaluate the selection.

'AQUA' then the executing script's path cannot be determined. Unlike 'Rgui', 'RStudio', 'VSCode', 'Jupyter', and 'Emacs', there is currently no way to request the path of an open document. Until such a time that there is a method for requesting the path of an open document, consider using another GUI.

If R is running in another manner, an error is thrown.

If your GUI of choice is not implemented with `this.path()`, please contact the package maintainer so that it can be implemented.

Value

character string.

Argument 'n'

By default, `this.path()` will look for a path based on the `srcref` of the call to `this.path()` and the environment in which `this.path()` was called. For example:

```
{
#line 1 "file1.R"
fun <- function() this.path::this.path(original = TRUE)
fun()
}
```

```
{
#line 1 "file2.R"
fun()
}
```

Both of these will return `"file1.R"` because that is where the call to `this.path()` is written.

But suppose we do not care where `this.path()` is called, but instead want to know where `fun()` is called. Pass argument `n = 1` to do so; `this.path()` will inspect the call and the calling environment one generation up the stack:

```
{
#line 1 "file1.R"
fun <- function() this.path::this.path(original = TRUE, n = 1)
fun()
}


{
#line 1 "file2.R"
fun()
}
```

These will return `"file1.R"` and `"file2.R"`, respectively, because those are where the calls to `fun()` are written.

But now suppose we wish to make a second function that uses `fun()`. We do not care where `fun()` is called, but instead want to know where `fun2()` is called. Add a formal argument `n = 0` to each function and pass `n = n + 1` to each sub-function:

```
{
#line 1 "file1.R"
fun <- function(n = 0) {
    this.path::this.path(original = TRUE, n = n + 1)
}
fun()
}


{
#line 1 "file2.R"
fun2 <- function(n = 0) fun(n = n + 1)
list(fun = fun(), fun2 = fun2())
}


{
#line 1 "file3.R"
fun3 <- function(n = 0) fun2(n = n + 1)
list(fun = fun(), fun2 = fun2(), fun3 = fun3())
}
```

Within each file, all these functions will return the path in which they are called, regardless of how deep `this.path()` is called.

Note

If you need to use `this.path()` inside the site-wide startup profile file or a user profile, please use `with_site.file()` or `with_init.file()`, something along the lines of:

```
this.path::with_init.file({
    <expr 1>
    <expr 2>
    <...>
})
```

See Also

`shFILE()`

`set.sys.path()`, `wrap.source()`

Examples

```
FILE1.R <- tempfile(fileext = ".R")
writeLines("writeLines(sQuote(this.path::this.path()))", FILE1.R)

## 'this.path()' works with 'source()'
source(FILE1.R)

## 'this.path()' works with 'sys.source()'
sys.source(FILE1.R, envir = environment())

## 'this.path()' works with 'debugSource()' in 'RStudio'
if (.Platform$GUI == "RStudio")
    get("debugSource", "tools:rstudio", inherits = FALSE)(FILE1.R)

## 'this.path()' works with 'testthat::source_file()'
if (requireNamespace("testthat"))
    testthat::source_file(FILE1.R, chdir = FALSE, wrap = FALSE)

## 'this.path()' works with 'compiler::loadcmp()'
if (requireNamespace("compiler")) {
    FILE2.Rc <- tempfile(fileext = ".Rc")
    compiler::cmpfile(FILE1.R, FILE2.Rc)
    compiler::loadcmp(FILE2.Rc)
    unlink(FILE2.Rc)
}

## 'this.path()' works with 'Rscript'
this.path:::.Rscript(c("--default-packages=NULL", "--vanilla", FILE1.R))

## 'this.path()' also works when 'source()'-ing a URL
## (included tryCatch in case an internet connection is not available)
tryCatch({
    source(paste0("https://raw.githubusercontent.com/ArcadeAntics/",
                  "this.path/main/tests/sys-path-with-urls.R"))
}, condition = this.path:::.cat.condition)

unlink(FILE1.R)
```

try.this.path *Attempt to Determine Script's Filename*

Description

try.this.path() attempts to return this.path(), returning this.path(original = TRUE) if that fails, returning NA_character_ if that fails as well.

Usage

```
try.this.path(contents = FALSE, local = FALSE, n = 0,
              envir = parent.frame(n + 1),
              matchThisEnv = getOption("topLevelEnvironment"),
              srcfile = if (n) sys.parent(n) else 0)

try.shFILE()
```

Arguments

contents, local, n, envir, matchThisEnv, srcfile
: See ?this.path().

Details

This should **NOT** be used to construct file paths against the script's directory. This should exclusively be used for diagnostic messages / / warnings / / errors / / logging. The returned path may not exist, may be relative instead of absolute, or may be undefined.

Value

character string.

Examples

```
try.shFILE()
try.this.path()
try.this.path(contents = TRUE)
```

tryCatch2 *Condition Handling and Recovery*

Description

Variants of tryCatch() that accept an else. argument, similar to try except in 'Python'.

last.condition will be the last thrown and caught condition in tryCatch3().

Usage

```
tryCatch2(expr, ..., else., finally)
tryCatch3(expr, ..., else., finally)

last.condition
```

Arguments

`expr`	expression to be evaluated.
`...`	for `tryCatch2()`, condition handlers. for `tryCatch3()`, expressions to be conditionally evaluated. Arguments which are missing will use the next non-missing argument. If there is no next non-missing argument, `NULL` will be returned invisibly.
`else.`	expression to be evaluated if evaluating `expr` does not throw an error nor a condition is caught.
`finally`	expression to be evaluated before returning or exiting.

Details

The use of the `else.` argument is better than adding additional code to `expr` because it avoids accidentally catching a condition that was not being protected by the `tryCatch()` call.

Examples

```
FILES <- tempfile(c("existent-file_", "non-existent-file_"))
writeLines("line1\nline2", FILES[[1L]])
for (FILE in FILES) {
    conn <- file(FILE)
    tryCatch2({
        open(conn, "r")
    }, condition = function(cond) {
        cat("cannot open", FILE, "\n")
    }, else. = {
        cat(FILE, "has", length(readLines(conn)), "lines\n")
    }, finally = {
        close(conn)
    })
    # ## or more naturely with tryCatch3:
    # tryCatch3({
    #     open(conn, "r")
    # }, condition = {
    #     cat("cannot open", FILE, "\n")
    # }, else. = {
    #     cat(FILE, "has", length(readLines(conn)), "lines\n")
    # }, finally = {
    #     close(conn)
    # })
}
unlink(FILES)
```

`wrap.source` *Implement 'this.path()' For Arbitrary 'source()'-Like Functions*

Description

`sys.path()` is implemented to work with `source()`, `sys.source()`, `debugSource()` in 'RStudio', `compiler::loadcmp()`, `box::use()`, `knitr::knit()`, `plumber::plumb()`, `shiny::runApp()`, **targets**, and `testthat::source_file()`.

`set.sys.path()` and `wrap.source()` can be used to implement `sys.path()` for any other `source()`-like functions.

`set.env.path()` and `set.src.path()` can be used along side `set.sys.path()` to implement `env.path()` and `src.path()`. Note that `set.env.path()` only makes sense if the code is being modularized, see examples.

`unset.sys.path()` will undo a call to `set.sys.path()`. You will need to use this if you wish to call `set.sys.path()` multiple times within a function.

`set.sys.path.function()` is a special variant of `set.sys.path()` to be called within `callr::r()` on a function with an appropriate `srcref`.

`with_sys.path()` is a convenient way to evaluate code within the context of a file. Whereas `set.sys.path()` can only be used within a function, `with_sys.path()` can only be used outside a function. `with_site.file()` and `with_init.file()` provide two common use cases of `with_sys.path()`; within the site-wide startup profile file or a user profile. You could combine these with `withAutoprint()` to auto-print expressions as if in a toplevel context, for example:

```
this.path::with_init.file(withAutoprint({
    this.path::this.path()
}, echo = FALSE))
```

See `?sys.path(local = TRUE)` which returns the path of the executing script, confining the search to the local environment in which `set.sys.path()` was called.

Usage

```
wrap.source(expr,
    path.only = FALSE,
    character.only = path.only,
    file.only = path.only,
    conv2utf8 = FALSE,
    allow.blank.string = FALSE,
    allow.clipboard = !file.only,
    allow.stdin = !file.only,
    allow.url = !file.only,
    allow.file.uri = !path.only,
    allow.unz = !path.only,
    allow.pipe = !file.only,
    allow.terminal = !file.only,
    allow.textConnection = !file.only,
    allow.rawConnection = !file.only,
    allow.sockconn = !file.only,
    allow.servsockconn = !file.only,
    allow.customConnection = !file.only,
    ignore.all = FALSE,
    ignore.blank.string = ignore.all,
    ignore.clipboard = ignore.all,
    ignore.stdin = ignore.all,
    ignore.url = ignore.all,
    ignore.file.uri = ignore.all)

set.sys.path(file,
    path.only = FALSE,
    character.only = path.only,
```

```
    file.only = path.only,
    conv2utf8 = FALSE,
    allow.blank.string = FALSE,
    allow.clipboard = !file.only,
    allow.stdin = !file.only,
    allow.url = !file.only,
    allow.file.uri = !path.only,
    allow.unz = !path.only,
    allow.pipe = !file.only,
    allow.terminal = !file.only,
    allow.textConnection = !file.only,
    allow.rawConnection = !file.only,
    allow.sockconn = !file.only,
    allow.servsockconn = !file.only,
    allow.customConnection = !file.only,
    ignore.all = FALSE,
    ignore.blank.string = ignore.all,
    ignore.clipboard = ignore.all,
    ignore.stdin = ignore.all,
    ignore.url = ignore.all,
    ignore.file.uri = ignore.all,
    Function = NULL, ofile)

set.env.path(envir, matchThisEnv = getOption("topLevelEnvironment"))

set.src.path(srcfile)

unset.sys.path()

set.sys.path.function(fun)

with_sys.path(file, expr, ...)
with_site.file(expr, n = 0)
with_init.file(expr, n = 0)
```

Arguments

expr	for `wrap.source()`, an (unevaluated) call to a `source()`-like function. for `with_sys.path()`, `with_site.file()`, and `with_init.file()`, an expression to evaluate within the context of a file.
file	a connection or a character string giving the pathname of the file or URL to read from.
path.only	must `file` be an existing path? This implies `character.only` and `file.only` are `TRUE` and implies `allow.file.uri` and `allow.unz` are `FALSE`, though these can be manually changed.
character.only	must `file` be a character string?
file.only	must `file` refer to an existing file?
conv2utf8	if `file` is a character string, should it be converted to UTF-8?
allow.blank.string	may `file` be a blank string, i.e. `""`?

`allow.clipboard`
: may `file` be `"clipboard"` or a clipboard connection?

`allow.stdin`
: may `file` be `"stdin"`? Note that `"stdin"` refers to the C-level 'standard input' of the process, differing from `stdin()` which refers to the R-level 'standard input'.

`allow.url`
: may `file` be a URL pathname or a connection of class `"url-libcurl"` / / `"url-wininet"`?

`allow.file.uri`
: may `file` be a '`file://`' URI?

`allow.unz, allow.pipe, allow.terminal, allow.textConnection, allow.rawConnection, allow.sockconn, a`
: may `file` be a connection of class `"unz"` / / `"pipe"` / / `"terminal"` / / `"textConnection"` / / `"rawConnection"` / / `"sockconn"` / / `"servsockconn"`?

`allow.customConnection`
: may `file` be a custom connection?

`ignore.all, ignore.blank.string, ignore.clipboard, ignore.stdin, ignore.url, ignore.file.uri`
: ignore the special meaning of these types of strings, treating it as a path instead?

`Function`
: character vector of length 1 or 2; the name of the function and package in which `set.sys.path()` is called.

`ofile`
: a connection or a character string specifying the original file argument. This overwrites the value returned by `sys.path(original = TRUE)`.

`envir, matchThisEnv`
: arguments passed to `topenv()` to determine the top level environment in which to assign an associated path.

`srcfile`
: source file in which to assign a pathname.

`fun`
: function with a `srcref`.

`...`
: further arguments passed to `set.sys.path()`.

`n`
: see `?this.path()`.

Details

`set.sys.path()` should be added to the body of your `source()`-like function before reading / / evaluating the expressions.

`wrap.source()`, unlike `set.sys.path()`, does not accept an argument `file`. Instead, an attempt is made to extract the file from `expr`, after which `expr` is evaluated. It is assumed that the file is the first argument of the function, as is the case with `source()`, `sys.source()`, `debugSource()` in 'RStudio', `compiler::loadcmp()`, `knitr::knit()`, and `testthat::source_file()`. The function of the call is evaluated, its `formals()` are retrieved, and then the arguments of `expr` are searched for a name matching the name of the first formal argument. If a match cannot be found by name, the first unnamed argument is taken instead. If no such argument exists, the file is assumed missing.

`wrap.source()` does non-standard evaluation and does some guess work to determine the file. As such, it is less desirable than `set.sys.path()` when the option is available. I can think of exactly one scenario in which `wrap.source()` might be preferable: suppose there is a `source()`-like function `sourcelike()` in a foreign package (a package for which you do not have write permission). Suppose that you write your own function in which the formals are `(...)` to wrap `sourcelike()`:

```
wrapper <- function (...)
{
    ## possibly more args to wrap.source()
    wrap.source(sourcelike(...))
}
```

This is the only scenario in which `wrap.source()` is preferable, since extracting the file from the `...` list would be a pain. Then again, you could simply change the formals of `wrapper()` from `(...)` to `(file, ...)`. If this does not describe your exact scenario, use `set.sys.path()` instead.

Value

for `wrap.source()`, the result of evaluating `expr`.

for `set.sys.path()`, if `file` is a path, then the normalized path with the same attributes, otherwise `file` itself. The return value of `set.sys.path()` should be assigned to a variable before use, something like:

```
{
    file <- set.sys.path(file, ...)
    sourcelike(file)
}
```

Using 'ofile'

`ofile` can be used when the `file` argument supplied to `set.sys.path()` is not the same as the `file` argument supplied to the `source()`-like function:

```
sourcelike <- function (file)
{
    ofile <- file
    if (!is.character(ofile) || length(ofile) != 1)
        stop(gettextf("'%s' must be a character string", "file"))
    ## if the file exists, do nothing
    if (file.exists(file)) {
    }
    ## look for the file in the home directory
    ## if it exists, do nothing
    else if (file.exists(file <- this.path::path.join("~", ofile))) {
    }
    ## you could add other directories to look in,
    ## but this is good enough for an example
    else stop(gettextf("'%s' is not an existing file", ofile))
    file <- this.path::set.sys.path(file, ofile = ofile)
    exprs <- parse(n = -1, file = file)
    for (i in seq_along(exprs)) eval(exprs[i], envir)
    invisible()
}
```

Note

Both functions should only be called within another function.

Suppose that the functions `source()`, `sys.source()`, `debugSource()` in 'RStudio', `compiler::loadcmp()`, `box::use()`, `knitr::knit()`, `plumber::plumb()`, `shiny::runApp()`, `targets`, and `testthat::source_file()` were not implemented with `sys.path()`. You could use `set.sys.path()` to implement each of them as follows:

```
source() wrapper <- function(file, ...) {
        file <- set.sys.path(file)
        source(file = file, ...)
    }
```

```
sys.source() wrapper <- function(file, ...) {
        file <- set.sys.path(file, path.only = TRUE)
        sys.source(file = file, ...)
    }
```

debugSource() **in 'RStudio'**

```
wrapper <- function(fileName, ...) {
        fileName <- set.sys.path(fileName, character.only = TRUE,
            conv2utf8 = TRUE, allow.blank.string = TRUE)
        debugSource(fileName = fileName, ...)
    }
compiler::loadcmp() wrapper <- function(file, ...) {
        file <- set.sys.path(file, path.only = TRUE)
        compiler::loadcmp(file = file, ...)
    }
knitr::knit() wrapper <- function(input, ...) {
        ## this works for the most part, but will not work in child mode
        input <- set.sys.path(input, allow.file.uri = FALSE)
        knitr::knit(input = input, ...)
    }
testthat::source_file() wrapper <- function(path, ...) {
        ## before testthat_3.1.2, source_file() used readLines() to read
        ## the input lines. changed in 3.1.2, source_file() uses
        ## brio::read_lines() which normalizes 'path' before reading,
        ## disregarding the special meaning of the strings listed above
        path <- set.sys.path(path, path.only = TRUE, ignore.all =
            as.numeric_version(getNamespaceVersion("testthat")) >= "3.1.2")
        testthat::source_file(path = path, ...)
    }
```

box::use(), plumber::plumb(), shiny::runApp(), and targets do not have any simple implementations using set.sys.path() since the sourcing functions are the internal objects of these namespaces.

Examples

```
FILE.R <- tempfile(fileext = ".R")
this.path:::.write.code({
    this.path::sys.path(verbose = TRUE)
    try(this.path::env.path(verbose = TRUE))
    this.path::src.path(verbose = TRUE)
    this.path::this.path(verbose = TRUE)
}, FILE.R)


## here we have a source-like function, suppose this
## function is in a package for which you have write permission
sourcelike <- function (file, envir = parent.frame())
{
    ofile <- file
    file <- set.sys.path(file, Function = "sourcelike")
    lines <- readLines(file, warn = FALSE)
    filename <- sys.path(local = TRUE, for.msg = TRUE)
    isFile <- !is.na(filename)
    if (isFile) {
        timestamp <- file.mtime(filename)[1]
```

```
        ## in case 'ofile' is a URL pathname / / 'unz' connection
        if (is.na(timestamp))
            timestamp <- Sys.time()
    }
    else {
        filename <- if (is.character(ofile)) ofile else "<connection>"
        timestamp <- Sys.time()
    }
    srcfile <- srcfilecopy(filename, lines, timestamp, isFile)
    set.src.path(srcfile)
    exprs <- parse(text = lines, srcfile = srcfile, keep.source = FALSE)
    invisible(source.exprs(exprs, evaluated = TRUE, envir = envir))
}


sourcelike(FILE.R)
sourcelike(conn <- file(FILE.R)); close(conn)


## here we have another source-like function, suppose this function
## is in a foreign package for which you do not have write permission
sourcelike2 <- function (pathname, envir = globalenv())
{
    if (!(is.character(pathname) && file.exists(pathname)))
        stop(gettextf("'%s' is not an existing file",
             pathname, domain = "R-base"))
    envir <- as.environment(envir)
    lines <- readLines(pathname, warn = FALSE)
    srcfile <- srcfilecopy(pathname, lines, isFile = TRUE)
    exprs <- parse(text = lines, srcfile = srcfile, keep.source = FALSE)
    invisible(source.exprs(exprs, evaluated = TRUE, envir = envir))
}


## the above function is similar to sys.source(), and it
## expects a character string referring to an existing file
##
## with the following, you should be able
## to use 'sys.path()' within 'FILE.R':
wrap.source(sourcelike2(FILE.R), path.only = TRUE)


# ## with R >= 4.1.0, use the forward pipe operator '|>' to
# ## make calls to 'wrap.source' more intuitive:
# sourcelike2(FILE.R) |> wrap.source(path.only = TRUE)


## 'wrap.source' can recognize arguments by name, so they
## do not need to appear in the same order as the formals
wrap.source(sourcelike2(envir = new.env(), pathname = FILE.R),
    path.only = TRUE)


## it it much easier to define a new function to do this
sourcelike3 <- function (...)
wrap.source(sourcelike2(...), path.only = TRUE)
```

```
## the same as before
sourcelike3(FILE.R)


## however, this is preferable:
sourcelike4 <- function (pathname, ...)
{
    ## pathname is now normalized
    pathname <- set.sys.path(pathname, path.only = TRUE)
    sourcelike2(pathname = pathname, ...)
}
sourcelike4(FILE.R)


## perhaps you wish to run several scripts in the same function
fun <- function (paths, ...)
{
    for (pathname in paths) {
        pathname <- set.sys.path(pathname, path.only = TRUE)
        sourcelike2(pathname = pathname, ...)
        unset.sys.path(pathname)
    }
}


## here we have a source-like function which modularizes its code
sourcelike5 <- function (file)
{
    ofile <- file
    file <- set.sys.path(file, Function = "sourcelike5")
    lines <- readLines(file, warn = FALSE)
    filename <- sys.path(local = TRUE, for.msg = TRUE)
    isFile <- !is.na(filename)
    if (isFile) {
        timestamp <- file.mtime(filename)[1]
        ## in case 'ofile' is a URL pathname / / 'unz' connection
        if (is.na(timestamp))
            timestamp <- Sys.time()
    }
    else {
        filename <- if (is.character(ofile)) ofile else "<connection>"
        timestamp <- Sys.time()
    }
    srcfile <- srcfilecopy(filename, lines, timestamp, isFile)
    set.src.path(srcfile)
    envir <- new.env(hash = TRUE, parent = .BaseNamespaceEnv)
    envir$.packageName <- filename
    oopt <- options(topLevelEnvironment = envir)
    on.exit(options(oopt))
    set.env.path(envir)
    exprs <- parse(text = lines, srcfile = srcfile, keep.source = FALSE)
    source.exprs(exprs, evaluated = TRUE, envir = envir)
    envir
}
```

```
sourcelike5(FILE.R)


## the code can be made much simpler in some cases
sourcelike6 <- function (file)
{
    ## we expect a character string refering to a file
    ofile <- file
    filename <- set.sys.path(file, path.only = TRUE, ignore.all = TRUE,
        Function = "sourcelike6")
    lines <- readLines(filename, warn = FALSE)
    timestamp <- file.mtime(filename)[1]
    srcfile <- srcfilecopy(filename, lines, timestamp, isFile = TRUE)
    set.src.path(srcfile)
    envir <- new.env(hash = TRUE, parent = .BaseNamespaceEnv)
    envir$.packageName <- filename
    oopt <- options(topLevelEnvironment = envir)
    on.exit(options(oopt))
    set.env.path(envir)
    exprs <- parse(text = lines, srcfile = srcfile, keep.source = FALSE)
    source.exprs(exprs, evaluated = TRUE, envir = envir)
    envir
}


sourcelike6(FILE.R)


unlink(FILE.R)
```

Index