Package 'this.path'

December 1, 2022

Version 1.1.0

License MIT + file LICENSE

Title Get Executing Script's Path, from 'RStudio', 'Rgui', 'VSCode', 'Rscript' (Shells Including Windows Command-Line / / Unix Terminal), and 'source'

Description Determine the full path of the executing script. Works when running a line or selection from a script in 'RStudio', 'Rgui', and 'VSCode', when using 'source', 'sys.source', 'debugSource' in 'RStudio', 'testthat::source_file', and 'knitr::knit', and when running from a shell.

Author Andrew Simmons

Maintainer Andrew Simmons <akwsimmo@gmail.com>

Suggests utils, microbenchmark

Enhances knitr, rprojroot, rstudioapi, testthat

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

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

Biarch TRUE

Encoding UTF-8

R topics documented:

`this.path-package` *Get Executing Script's Path, from 'RStudio', 'Rgui', 'VSCode', 'Rscript' (Shells Including Windows Command-Line / / Unix Terminal), and 'source'*

Description

Determine the full path of the executing script. Works when running a line or selection from a script in 'RStudio', 'Rgui', and 'VSCode', when using 'source', 'sys.source', 'debugSource' in 'RStudio', 'testthat::source_file', and 'knitr::knit', and when running from a shell.

Details

`this.path()` returns the `normalized` path of the executing script.

`this.dir()` returns the `normalized` path of the directory in which the executing script is located.

`here()` constructs file paths relative to the executing script's directory.

`path.join()`, `basename2()`, and `dirname2()` are replacements for `file.path()`, `basename()`, and `dirname()` that better handle drives, network shares, and a few important edge cases.

`splitext()`, `removeext()`, `ext()`, and `ext<-()` split paths into root and extension, remove extensions, get extensions, and set extensions, respectively.

`check.path()` and `check.dir()` check that `this.path()` and `this.dir()` are functioning correctly.

`as.rel.path()` (or `as.relative.path()`) will turn absolute paths into relative paths.

`asArgs()`, `fileArgs()`, `progArgs()`, and `withArgs()` provide functionality for running scripts with arguments in the same session, as opposed to a new one with `Rscript`.

`is.main()` and `from.shell()` determine if an R script is the main executing script or is run from a shell.

`shFILE()` and `normalized.shFILE()` extract 'FILE' from the command line arguments.

`tryCatch2()` adds argument `else.` that runs if no error is thrown. This helps to run extra code that is not intended to be protected by the condition handlers.

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)

Andrew Simmons

Maintainer: Andrew Simmons <akwsimmo@gmail.com>

See Also

`source`, `sys.source`, `debugSource` in 'RStudio', `testthat::source_file`, `knitr::knit`

`R.from.shell`

`Args` *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 scripts 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', `testthat::source_file`, or `knitr::knit`.

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 assocaited 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", )` (can easily convert 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 `expr`.

Examples

```
this.path::asArgs(
    NULL,
    c(TRUE, FALSE, NA),
    1:5,
    pi,
    exp(6i),
    letters[1:5],
    as.raw(0:4),
    as.Date("1970-01-01"),
    as.POSIXct("1970-01-01 00:00:00"),
    list(
        list(
            list(
                "lists are recursed"
            )
        )
    )
)


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


# wrap your source call with a call to 'withArgs'
this.path::withArgs(
    source(FILE, local = TRUE, verbose = FALSE),
    letters, pi, exp(1)
)
this.path::withArgs(
    sys.source(FILE, environment()),
    letters, pi + 1i * exp(1)
)
this.path:::.Rscript(c("--default-packages=this.path", "--vanilla", FILE,
    this.path::asArgs(letters, pi, as.POSIXct("2022-07-17 04:25"))))
```

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


# withArgs() also works with inside.source() and wrap.source()
sourcelike <- function (file, envir = parent.frame())
{
    file <- inside.source(file)
    envir <- as.environment(envir)
    exprs <- parse(n = -1, file = file, srcfile = NULL, keep.source = FALSE)
    for (i in seq_along(exprs)) eval(exprs[i], envir)
}
this.path::withArgs(sourcelike(FILE), letters)


sourcelike2 <- function (file, envir = parent.frame())
{
    envir <- as.environment(envir)
    exprs <- parse(n = -1, file = file, srcfile = NULL, keep.source = FALSE)
    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), letters)
this.path::withArgs(wrap.source(sourcelike2(FILE)), letters)
```

`as.relative.path` *Make a Path Relative to Another Path*

Description

When working with **this.path**, you will be dealing with a lot of absolute paths. These paths are no good for saving within files, so you'll need to use `as.relative.path()` to turn your absolute paths into relative paths.

Usage

```
as.relative.path(path, relative.to = this.dir(verbose = FALSE))
as.rel.path     (path, relative.to = this.dir(verbose = FALSE))
relpath         (path, relative.to = getwd())
```

Arguments

path	character vector of file / / URL paths.
relative.to	character string; the file / / URL path to make path relative to.

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.

Note

relpath and as.rel.path are the same function with different default arguments.

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
      "https://raw.githubusercontent.com/ArcadeAntics/this.path/main/tests/this.path_w_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/this.path.R"
    ),
    relative.to = "C:/Users/effective_user/Documents/this.path/man"
)

## End(Not run)
```

---

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 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.

And lastly, 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 begin with "/".

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

Behaviour Elsewhere

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", "/", ".", "..")
print(cbind(
    path, dirname = dirname2(path), basename = basename2(path)),
    quote = FALSE, print.gap = 3)
```

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

Description

Add check.path("path/to/file") to the beginning of your script to initialize this.path(), and check that this.path() is returning the path you expect.

Usage

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

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 using an absolute path is not intended (recommended against). this.path() makes R scripts portable, but using an absolute path in check.path or check.dir makes an R script non-portable, defeating the whole purpose of this package.

Value

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

Otherwise, an error is raised.

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` of the path will be performed.

Trailing path separators are removed before dissecting the path.

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")

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

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

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

`from.shell` *Top-Level Code Environment*

Description

Determine is 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(file = FILES[1], bquote(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+ ")))
this.path:::write.code({
    from.shell()
    is.main()
}, FILES[2])


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], verbose = FALSE)\\n\")",
    "-e", "source(commandArgs(TRUE)[1L], verbose = FALSE)",
    FILES[1]))
```

`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 working directory is not available.

Examples

```
cat("initial working directory:\n")
getinitwd()

cat("current working directory:\n")
getwd()
```

`here` *Construct Path to File, Beginning with* `this.dir()`

Description

Construct the path to a file from components/paths in a platform-**DEPENDENT** way, starting with `this.dir()`.

Usage

```
here(..., .. = 0)
ici(..., .. = 0)
```

Arguments

`...`	further arguments passed to `path.join()`.
`..`	the number of directories to go back.

Details

The path to a file begins with a base. The base is `..` number of directories back from the executing script's directory (`this.dir()`). The argument is named `..` because `".."` refers to the parent directory in Windows, Unix, and URL paths alike.

Value

A character vector of the arguments concatenated term-by-term, beginning with the executing script's directory.

Examples

```
FILE <- tempfile(fileext = ".R")
this.path:::write.code({


    this.path::here()
    this.path::here(.. = 1)
    this.path::here(.. = 2)


    # use 'here' to read input from a file located nearby
    this.path::here(.. = 1, "input", "file1.csv")


    # or maybe to run another script
    this.path::here("script2.R")


}, FILE)


source(FILE, echo = TRUE, verbose = FALSE)
```

LINENO *Line Number of Executing Script*

Description

Get the line number of the executing script.

Usage

```
LINENO()
```

Value

An integer, `NA_integer_` if the line number cannot be determined.

Note

`LINENO()` only works if the executing script has a `srcref` and a `srcfile`. Scripts run with `Rscript` do not store their `srcref`, even when `getOption("keep.source")` is `TRUE`.

For `source` or `sys.source`, make sure to supply argument `keep.source = TRUE` directly, or set the options `"keep.source"` or `"keep.source.pkgs"` to `TRUE`.

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

For testthat::source_file, the srcref is always stored, so you do not need to do anything special before calling.

For knitr::knit, there is nothing that can be done, the srcref is never stored. I'm looking into a fix for such a thing.

Examples

```
FILE <- tempfile(fileext = ".R")
writeLines(c(
    "LINENO()",
    "LINENO()",
    "# LINENO() respects #line directives",
    "#line 1218",
    "LINENO()"
), FILE)
# source(FILE, echo = TRUE, verbose = FALSE,
#     max.deparse.length = Inf, keep.source = TRUE)
#
# 'source(echo = TRUE, keep.source = TRUE)'
# echoes incorrectly with #line directives
#
# 'source2()' echoes correctly!
this.path:::source2(FILE, echo = TRUE, verbose = FALSE,
    max.deparse.length = Inf, keep.source = TRUE)
```

OS.type *Detect the Operating System Type*

Description

OS.type is a list of TRUE/FALSE values dependent of 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.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 paths 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")
```

R.from.shell *Using R From a Shell*

Description

How to use R from a shell (including the Windows command-line / / Unix terminal).

Details

For the purpose of running R scripts, there are four ways to do it. Suppose our R script has filename 'script1.R', we could write any of:

- R -f script1.R
- R --file=script1.R
- R CMD BATCH script1.R
- Rscript script1.R

The first two are different ways of writing equivalent statements. The third statement is the first statement plus options '--restore' '--save' (plus option '--no-readline' under Unix-alikes), and it also saves the stdout and stderr in a file of your choosing. The fourth statement is the second statement plus options '--no-echo' '--no-restore'. You can try:

- R --help
- R CMD BATCH --help
- Rscript --help

for a help message that describes what these options mean. In general, Rscript is the one you want to use. It should be noted that Rscript has some exclusive environment variables (not used by the other executables) that will make its behaviour different from R.

For the purpose of making packages, R CMD is what you'll need to use. Most commonly, you'll use:

- R CMD build
- R CMD INSTALL
- R CMD check

R CMD build will turn an R package (specified by a directory) into tarball. This allows for easy sharing of R packages with other people, including submitting a package to CRAN. R CMD INSTALL will install an R package (specified by a directory or tarball), and is used by utils::install.packages. R CMD check will check an R package (specified by a tarball) for possible errors in code, documentation, tests, and much more.

If, when you execute one of the previous commands, you see the following error message: "'R' is not recognized as an internal or external command, operable program or batch file.", see section **Ease of Use on Windows**.

Ease of Use on Windows

Under Unix-alikes, it is easy to invoke an R session from a shell by typing the name of the R executable you wish to run. On Windows, you should see that typing the name of the R executable you wish to run does not run that application, but instead signals an error. Instead, you will have to type the full path of the directory where your R executables are located (see section **Where are my R executable files located?**), followed by the name of the R executable you wish to run.

This is not very convenient to type everytime something needs to be run from a shell, plus it has another issue of being computer dependent. The solution is to add the path of the directory where your R executables are located to the `Path` environment variable. The `Path` environment variable is a list of directories where executable programs are located. When you type the name of an executable program you wish to run, Windows looks for that program through each directory in the `Path` environment variable. When you add the full path of the directory where your R executables are located to your `Path` environment variable, you should be able to run any of those executable programs by their basenames ('R', 'Rcmd', 'Rscript', and 'Rterm') instead of their full paths.

To add a new path to your `Path` environment variable:

1. Open the **Control Panel**
2. Open category **User Accounts**
3. Open category **User Accounts** (again)
4. Open **Change my environment variables**
5. Click the variable `Path`
6. Click the button **Edit...**
7. Click the button **New**
8. Type (or paste) the full path of the directory where your R executables are located, and press **OK**

This will modify your environment variable `Path`, not the systems. If another user wishes to run R from a shell, they will have to add the directory to their `Path` environment variable as well.

If you wish to modify the system environment variable `Path` (you will need admin permissions):

1. Open the **Control Panel**
2. Open category **System and Security**
3. Open category **System**
4. Open **Advanced system settings**
5. Click the button **Environment Variables...**
6. Modify `Path` same as before, just select `Path` in **System variables** instead of **User variables**

To check that this worked correctly, open a shell and execute the following commands:

- `R --help`
- `R --version`

You should see that the first prints the usage message for the R executable while the second prints information about the version of R currently being run. If you have multiple versions of R installed, make sure this is the version of R you wish to run.

Where are my R executable files located?

In an R session, you can find the location of your R executable files with the following command:
`cat(sQuote(normalizePath(R.home("bin"))), "\n")`

For me, this is:

`'C:\Program Files\R\R-4.2.2\bin\x64'`

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

Description

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

Usage

```
shFILE(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 if not.
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 the command line argument 'FILE' was not found.

Note

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

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

See Also

this.path, here

Examples

```
FILE <- tempfile(fileext = ".R")
this.path:::write.code({
    withAutoprint({
        shFILE(original = TRUE)
        shFILE()
        shFILE(default = {
            stop("interestingly enough, because 'FILE' will be found,\n",
                 "  argument 'default' won't be evaluated, and so this\n",
                 "  error won't actually print, isn't that neat? you can\n",
                 "  use this to your advantage in a similar manner, doing\n",
                 "  arbitrary things only if 'FILE' isn't found")
        })
    }, spaced = TRUE, verbose = FALSE, width.cutoff = 60L)
```

```
}, FILE)
this.path:::.Rscript(c("--default-packages=this.path", "--vanilla", FILE))


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))
}
```

`this.path` *Determine Executing Script's Filename*

Description

`this.path()` returns the `normalized` path of the executing script.

`this.dir()` returns the `normalized` path of the directory in which the executing script is located.

See also `here()` for constructing paths to files, starting with `this.dir()`.

`Sys.path()` and `Sys.dir()` are versions of `this.path()` and `this.dir()` that takes no arguments.

Usage

```
this.path(verbose = getOption("verbose"), original = FALSE,
    for.msg = FALSE, default, else.)
this.dir (verbose = getOption("verbose"), default, else.)

Sys.path()  # short for 'this.path(verbose = FALSE)'
Sys.dir ()  # short for 'this.dir (verbose = FALSE)'
```

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 if not.
`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've be thrown. `for.msg = TRUE` will ignore `original = FALSE`, and will use `original = NA` instead.
`default`	if there is no executing script, this value is returned.
`else.`	missing or a function to apply if there is an executing script. See `tryCatch2` for inspiration.

Details

There are three ways in which R code is typically run:

1. in 'RStudio' / / 'Rgui' / / 'VSCode' by running the current line / / selection with the **Run** button / / appropriate keyboard shortcut
2. through a source call: a call to function `source`, `sys.source`, `debugSource` in 'RStudio', `testthat::source_file`, or `knitr::knit`
3. from a shell, such as the Windows command-line / / Unix terminal

If you are using `this.path` in 'VSCode', see `this.path.in.VSCode`

To retrieve the executing script's filename, first an attempt is made to find a source call. The calls are searched in reverse order so as to grab the most recent source call in the case of nested source calls. If a source call was found, the argument *file* (*fileName* in the case of `debugSource`, *path* in the case of `testthat::source_file`, *input* in the case of `knitr::knit`) is returned from the function's evaluation environment. If you have your own source-like function that you'd like to be recognized by `this.path`, please contact the package maintainer so that it can be implemented.

If no source call is found up the calling stack, then an attempt is made to figure out how R is currently being used.

If R is being run from a shell, the shell arguments are searched for '`-f`' '`FILE`' or '`--file=FILE`' (the two methods of taking input from '`FILE`'). If exactly one of either type of argument is supplied, the text '`FILE`' is returned. It is an error to use `this.path` when none or multiple arguments of either type are supplied.

If R is being run from a shell under Unix-alikes with '`-g`' '`Tk`' or '`--gui=Tk`', `this.path()` will signal an error. This is because 'Tk' does not make use of its '`-f`' '`FILE`', '`--file=FILE`' argument.

If R is being run from 'RStudio', 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). 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. It is an error for no documents to be open or for a document to not exist (not saved anywhere).

If R is being run 'VSCode', 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. It is an error for a document to not exist (not saved anywhere).

If R is being run from 'Rgui', the source document's filename (the document most recently interacted with besides the R Console) 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. It is an error for no documents to be open or for a document to not exist (not saved anywhere).

If R is being run from 'AQUA', the executing script's path cannot be determined. Unlike 'RStudio' and 'Rgui', 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 'RStudio' or 'VSCode'.

If R is being run in another manner, it is an error to use `this.path`.

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; the executing script's filename.

Note

The first time `this.path` is called within a script, it will `normalize` the script's path, check that the script exists (throwing an error if it does not), and save it in the appropriate environment. When `this.path` is called subsequent times within the same script, it returns the saved path. This will be faster than the first time, will not check for file existence, and will be independent of the working directory.

As a side effect, this means that a script can delete itself using `file.remove` or `unlink` but still know its own path for the remainder of the script.

Within a script that contains calls to both `this.path` and `setwd`, `this.path` *MUST* be used *AT LEAST* once before the first call to `setwd`. This isn't always necessary; for instance if you ran a script using its absolute path as opposed to its relative path, changing the working directory has no effect. However, it is still advised against.

The following is *NOT* an example of bad practice:

```
setwd(this.path::this.dir())
```

`setwd` is most certainly written before `this.path()`, but `this.path()` will be evaluated first. It is not the written order that is bad practice, but the order of evaluation. Do not change the working directory before calling `this.path` at least once.

See Also

`here`

`shFILE`

`wrap.source`, `inside.source`

`this.path-package`

`source`, `sys.source`, `debugSource` in 'RStudio', `testthat::source_file`, `knitr::knit`

`R.from.shell`

Examples

```
FILE <- tempfile(fileext = ".R")
this.path:::write.code({
    withAutoprint({
        cat(sQuote(this.path::this.path(verbose = TRUE, default = {
            stop("interestingly enough, because the executing script's\n",
                "  path will be found, argument 'default' won't be evaluated,\n",
                "  and so this error won't actually print, isn't that\n",
                "  neat? you can use this to your advantage in a similar\n",
                "  manner, doing arbitrary things only if the executing\n",
                "  script does not exist")
        })), "\n\n")
    }, spaced = TRUE, verbose = FALSE, width.cutoff = 60L)
}, FILE)


source(FILE, verbose = FALSE)
sys.source(FILE, envir = environment())
if (.Platform$GUI == "RStudio")
```

```
    get("debugSource", "tools:rstudio", inherits = FALSE)(FILE)
if (requireNamespace("testthat"))
    testthat::source_file(FILE, chdir = FALSE, wrap = FALSE)
if (requireNamespace("knitr")) {
    writeLines(con = FILE2 <- tempfile(fileext = ".Rmd"), c(
        "```{r}",
        # same expression as above
        deparse(parse(FILE)[[c(1L, 2L, 2L)]]),
        "```"
    ))
    knitr::knit(input = FILE2, output = FILE3 <- tempfile(fileext = ".md"),
        quiet = TRUE)
    cat(sprintf("\n$ cat -bsT %s\n", shQuote(FILE2)))
    this.path:::cat.file(FILE2, number.nonblank = TRUE,
        squeeze.blank = TRUE, show.tabs = TRUE)
    cat(sprintf("\n$ cat -bsT %s\n", shQuote(FILE3)))
    this.path:::cat.file(FILE3, number.nonblank = TRUE,
        squeeze.blank = TRUE, show.tabs = TRUE)
    cat("\n")
    unlink(c(FILE3, FILE2))
}


this.path::::.Rscript(c("--default-packages=NULL", "--vanilla", FILE))


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


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


# an example from R package 'logr'
this.path::this.path(verbose = FALSE, default = "script.log",
    else. = function(path) {
        # replace extension (probably .R) with .log
        this.path::ext(path) <- ".log"
        path
        # or you could use paste0(this.path::removeext(path), ".log")
    })
```

---

this.path-defunct *Defunct Functions in Package* **this.path**

Description

The functions or variables listed here are no longer part of **this.path** as they are no longer needed.

Usage

```
# Defunct in 1.1.0
this.path2(...)         # use 'this.path(..., default = NULL)' instead
this.dir2(...)          # use 'this.dir(..., default = NULL)' instead
this.dir3(...)          # use 'this.dir(..., default = getwd())' instead
normalized.shFILE(...)  # use 'shFILE()' instead
```

Arguments

```
...
```

See Also

this.path, this.dir

---

this.path.in.VSCode this.path() *in 'VSCode'*

Description

this.path() will not work with a fresh installation of 'VSCode', some other packages / / applications are needed.

Details

You will need:

1. to install R packages **jsonlite** and **rlang**. On platforms without pre-built binaries, these packages will need compilation, you need to have make, gcc, and g++ installed. They can be installed from the terminal like:
   - sudo apt install make
   - sudo apt install gcc
   - sudo apt install g++
2. to install R package **rstudioapi**. This package does not require compilation. Note that you do not need to install 'RStudio'.
3. to install the R extension for 'VSCode'; a prompt to install it will appear upon opening an R script in 'VSCode'.

Furthermore, you must ensure that 'VSCode' has been attached properly. If you look at the bar on the bottom of your 'VSCode' window and see this:

R: (not attached)

then 'VSCode' is not attached to your R session and this.path() will fail. Click the button **R: (not attached)** and 'VSCode' will attempt to attach itself. If clicking the button throws this error:

Error in .vsc.attach() : could not find function ".vsc.attach"

then 'VSCode' was unable to attach itself, so you will need to restart your R session. Hover your cursor over **R Interactive** on the right side and click the trash bin labelled **Kill (Delete)** to kill your R session. After closing all of your R sessions, click the button **R: (not attached)** and 'VSCode' should now attach itself successfully. You should see something like this:

R 4.2.2

in the place of **R: (not attached)**.

`this.proj` *Construct Path to File, Beginning with Your Project Directory*

Description

`this.proj` behaves very similarly to `here::here` except that you can have multiple projects in use at once, and it will choose which project directory is appropriate based on `this.dir()`. Arguably, this makes it better than `here::here`.

Usage

```
this.proj(...)
```

Arguments

`...`	further arguments passed to `path.join()`.

Value

A character vector of the arguments concatenated term-by-term, beginning with the project directory.

See Also

`here`

`tryCatch2` *Condition Handling and Recovery*

Description

A variant of `tryCatch` that accepts an `else.` argument, similar to `try except` in 'Python'.

Usage

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

Arguments

`expr`	expression to be evaluated.
`...`	condition handlers.
`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 wasn't being protected by the `tryCatch` call.

Examples

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

---

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

Description

A source-like function is any function which evaluates code from a file.

Currently, `this.path()` is implemented to work with `source`, `sys.source`, `debugSource` in 'RStudio', `testthat::source_file`, and `knitr::knit`.

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

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)

inside.source(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)
```

Arguments

`expr`	an (unevaluated) call to a `source`-like function.
`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?

Details

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

`wrap.source()`, unlike `inside.source()`, 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', `testthat::source_file`, and `knitr::knit`. 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 `inside.source()` 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 `inside.source()` instead.

Value

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

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

```
{
    file <- inside.source(file, ...)
    sourcelike(file)
}
```

Note

Both functions should only be called within another function.

Suppose that the functions `source`, `sys.source`, `debugSource` in 'RStudio', `testthat::source_file`, and `knitr::knit` were not implemented with `this.path()`. You could use `inside.source()` to implement each of them as follows:

```
source wrapper <- function(file, ...) {
        file <- inside.source(file)
        source(file = file, ...)
    }
sys.source wrapper <- function(file, ...) {
        file <- inside.source(file, path.only = TRUE)
        sys.source(file = file, ...)
    }
debugSource in 'RStudio' wrapper <- function(fileName, ...) {
        fileName <- inside.source(fileName, character.only = TRUE,
            conv2utf8 = TRUE, allow.blank.string = TRUE)
        debugSource(fileName = fileName, ...)
    }
testthat::source_file wrapper <- function(path, ...) {
        # before testthat_3.1.2, source_file() used base::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 <- inside.source(path, path.only = TRUE,
          ignore.all = as.numeric_version(getNamespaceVersion("testthat")) >= "3.1.2")
        testthat::source_file(path = path, ...)
    }
knitr::knit wrapper <- function(input, ...) {
        # this works for the most part, but won't work in child mode
        input <- inside.source(input)
        knitr::knit(input = input, ...)
    }
```

Examples

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


# 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())
{
    file <- inside.source(file)
    envir <- as.environment(envir)
    exprs <- parse(n = -1, file = file, srcfile = NULL, keep.source = FALSE)
    # this prints nicely
    source(local = envir, echo = TRUE, exprs = exprs,
        spaced = TRUE, verbose = FALSE, max.deparse.length = Inf)
```

```
    # you could alternatively do:
    # 'for (i in seq_along(exprs)) eval(exprs[i], envir)'
    # which does no pretty printing
}


sourcelike(FILE)
sourcelike(con <- file(FILE)); close(con)


# 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)
    exprs <- parse(n = -1, file = pathname, srcfile = NULL, keep.source = FALSE)
    source(local = envir, echo = TRUE, exprs = exprs,
        spaced = TRUE, verbose = FALSE, max.deparse.length = Inf)
}


# 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 'this.path()' within 'FILE':
wrap.source(sourcelike2(FILE), path.only = TRUE)


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


# however, this is preferable:
sourcelike4 <- function (pathname, ...)
{
    # pathname is now normalized
    pathname <- inside.source(pathname, path.only = TRUE)
    sourcelike2(pathname = pathname, ...)
}
```

```
sourcelike4(FILE)
```

Index