Package 'this.path'

August 8, 2023

Version 2.0.0

Date 2023-08-08

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', and 'Rscript' (shell). Compatible with several functions and packages: 'source()', 'sys.source()', 'debugSource()' in 'RStudio', 'testthat::source_file()', 'knitr::knit()', 'compiler::loadcmp()', and 'box::use()'.

Author Iris Simmons

Maintainer Iris Simmons <ikwsimmo@gmail.com>

Suggests utils, microbenchmark

Enhances compiler, box, IRkernel, jsonlite, knitr, rprojroot, rstudioapi, 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', and '`Rscript`' (shell). Compatible with several functions and packages: `source()`, `sys.source()`, `debugSource()` in 'RStudio', `testthat::source_file()`, `knitr::knit()`, `compiler::loadcmp()`, and `box::use()`.

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:

- `LINENO()` returns the line number of the executing expression.
- `wrap.source()`, `set.sys.path()`, and `unset.sys.path()` implement `this.path()` for any `source()`-like functions outside of `source()`, `sys.source()`, `debugSource()` in 'RStudio', `testthat::source_file()`, `knitr::knit()`, `compiler::loadcmp()`, and `box::use()`.
- `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 *Active Bindings in Package* **this.path**

Description

These variables are active bindings, meaning they call functions when their values are requested.

`FILE` links to `try.this.path()`.

`LINE` links to `LINENO()`.

Usage

```
FILE
LINE
```

Details

These are intended to be used in a similar manner to the macros `__FILE__` and `__LINE__` in C. They are useful for generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected.

Examples

```
FILE.R <- tempfile(fileext = ".R")
writeLines("fun <- function ()
{
    message(sprintf('invalid value %d at %s, line %d',
        ## do not forget the braces around 'FILE' and 'LINE'
        ## see section Note of ?this.path for more details
        -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

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

Usage

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

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

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

Arguments

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

Details

The path to a file starts 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 on Windows, under Unix-alikes, and for URL pathnames.

Value

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

Examples

```
FILE.R <- 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.R)


source(FILE.R, echo = TRUE, verbose = FALSE)


unlink(FILE.R)
```

LINENO *Line Number of Executing Expression*

Description

Get the line number of the executing expression.

Usage

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

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

Arguments

n	See ?this.path().
envir, matchThisEnv	See ?env.path().
srcfile	See ?src.path().

Details

sys.LINENO() returns the line number of the most recent expression with a source reference and a source file equal to sys.path().

env.LINENO() returns the line number of the most recent expression with a source reference and a source file equal to env.path().

src.LINENO() returns the line number of its source file.

LINENO() returns the line number of the most recent expression with a source reference and a source file equal to this.path().

In general, LINENO() is the most useful. It works whether your R code is source()-d or modularized.

Value

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

Note

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

As mentioned in section **Note** of `?this.path`, source references are stored only for top-level expressions (including directly inside braces). For example:

```
#line 1
if (TRUE)
    message(sprintf("LINENO() will be 1: %d",
                    LINENO()))

#line 1
if (TRUE) {
    message(sprintf("LINENO() will be 2 because of the brace on line 1: %d",
                    LINENO()))
}

#line 1
if (TRUE) {
    message(sprintf("LINENO() will be 3 because of the braces on line 3: %d",
                    { LINENO() }))
}
```

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()`, the `srcref` is never stored, there is nothing that can be done. I am looking into a fix for such a thing.

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

For `box::use()`, the `srcref` is always stored, so you do not need to do anything special before calling.

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 = sys.call(if (n) sys.parent(n) else 0))
```

Arguments

`file`	a character string giving the pathname of the file or URL.
`local`	See ?`sys.path()`.
`n`	See ?`this.path()`.
`envir, matchThisEnv`	See ?`env.path()`.
`srcfile`	See ?`src.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 with formals `NULL` 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's root directory.
`rel2here, rel2proj`	Functions with formals `(path)` which turn absolute paths into relative paths, against the file's directory / / project's root directory.
`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))
```

---

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', `testthat::source_file()`, `knitr::knit()`, `compiler::loadcmp()`, or a `source()`-like function containing `wrap.source()` // `set.sys.path()`.

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 Path*

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))

rel2sys.dir(path, local = FALSE)
rel2sys.proj(path, local = FALSE)

rel2env.dir(path, n = 0, envir = parent.frame(n + 1),
            matchThisEnv = getOption("topLevelEnvironment"))
rel2env.proj(path, n = 0, envir = parent.frame(n + 1),
             matchThisEnv = getOption("topLevelEnvironment"))

rel2src.dir(path, n = 0,
            srcfile = sys.call(if (n) sys.parent(n) else 0))
rel2src.proj(path, n = 0,
             srcfile = sys.call(if (n) sys.parent(n) else 0))

rel2here(path, local = FALSE, n = 0, envir = parent.frame(n + 1),
         matchThisEnv = getOption("topLevelEnvironment"),
         srcfile = sys.call(if (n) sys.parent(n) else 0))
rel2proj(path, local = FALSE, n = 0,
         envir = parent.frame(n + 1),
         matchThisEnv = getOption("topLevelEnvironment"),
         srcfile = sys.call(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`	See ?`sys.path()`.
`n`	See ?`this.path()`.
`envir, matchThisEnv`	See ?`env.path()`.
`srcfile`	See ?`src.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.

Note

rel2sys.dir(), rel2sys.proj(), rel2env.dir(), rel2env.proj(), rel2src.dir(), rel2src.proj(), rel2here(), and rel2proj() are variants of relpath() in which relative.to is sys.dir(), sys.proj(), env.dir(), env.proj(), src.dir(), src.proj(), here(), and this.proj().

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.sys.path.jupyter *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.sys.path.jupyter() to declare the executing 'Jupyter' notebook's filename.

Usage

```
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.sys.path.jupyter("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.)
```

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 *Evaluating and Print Expressions*

Description

source.exprs() evaluates and auto-prints expression as if in a toplevel contenxt.

Usage

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

Arguments

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

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

Description

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

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

Usage

```
sys.path(verbose = getOption("verbose"), original = FALSE,
         for.msg = FALSE, contents = FALSE, local = FALSE,
         default, else.)
sys.dir(verbose = getOption("verbose"), local = FALSE,
        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 executing 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(sys.path(contents = TRUE)))` if you require a character vector. This is intended for logging purposes. This is useful in 'RStudio' and 'VSCode' 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?
`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 'Rgui' / / 'RStudio' / / 'VSCode' / / 'Jupyter' 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()`, `knitr::knit()`, `compiler::loadcmp()`, or `box::use()`

3. from a shell, such as the Windows command-line / / Unix terminal

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 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 `sys.path()`, please contact the package maintainer so that it can be implemented or use `wrap.source()` / / `set.sys.path()`.

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

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`'). The last '`FILE`' is extracted and returned (ignoring '`-f`' '`-`' and '`--file=-`'). It is an error to use `sys.path()` if no arguments of either type are supplied.

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

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 `sys.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 '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). 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 `sys.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 '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 `sys.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 'Jupyter', 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 `sys.path()`, or explicitly use `set.sys.path.jupyter()`.

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

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

Value

character string; the executing script's filename.

Note

The first time `sys.path()` is called within a script, it will normalize the script's path, checking that the script exists (throwing an error if it does not), and save it in the appropriate environment. When `sys.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.

Please **DO NOT** use `sys.path()` inside the site-wide startup profile file, the user profile, nor the function `.First()` (see `?Startup`). This has inconsistent results dependent on the GUI, and often incorrect. For example:

in 'Rterm' in all three cases, it returns '`FILE`' from the command line arguments:

```
> sys.path(original = TRUE)
Source: shell argument 'FILE'
[1] "./file569c63d647ba.R"

> sys.path()
[1] "C:/Users/iris/AppData/Local/Temp/RtmpGMmR3A/file569c63d647ba.R"
```

in 'Rgui' in all three cases, it throws an error:

```
> sys.path(original = TRUE)
Error in .sys.path.toplevel(FALSE, TRUE) :
  R is being run from Rgui with no documents open
```

in 'RStudio' in all three cases, it throws an error:

```
> sys.path(original = TRUE)
Error in .rs.api.getSourceEditorContext() :
  RStudio has not finished loading
```

in 'VSCode' in the site-wide startup profile file and the function `.First()`, it throws an error:

```
> sys.path(original = TRUE)
Error : RStudio not running
```

but in the user profile, it returns:

```
> sys.path(original = TRUE)
Source: call to function source
[1] "~/.Rprofile"

> sys.path()
[1] "C:/Users/iris/Documents/.Rprofile"
```

in 'Jupyter' in all three cases, it throws an error:

```
> sys.path(original = TRUE)
Error in .sys.path.toplevel(FALSE, TRUE) :
  Jupyter has not finished loading
```

Sometimes it returns '`FILE`' from the command line arguments, sometimes it returns the path of the user profile, and other times it throws an error. Alternatively, you could use `shFILE()`, supplying a `default` argument when no '`FILE`' is specificed, and supplying an `else.` function for when one is specified.

See Also

shFILE()

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

Examples

```
FILE1.R <- tempfile(fileext = ".R")
this.path:::.write.code({
    this.path:::.withAutoprint({
        cat(sQuote(this.path::sys.path(verbose = TRUE, default = {
            stop("since the executing script's path will be found,\n",
                " argument 'default' will not be evaluated, so this\n",
                " error will not be thrown! you can use this to\n",
                " your advantage in a similar manner, doing\n",
                " arbitrary things only if the executing script\n",
                " does not exist")
        })), "\n\n")
    }, spaced = TRUE, verbose = FALSE, width.cutoff = 58L,
       prompt = Sys.getenv("R_PROMPT"), continue = Sys.getenv("R_CONTINUE"))
}, FILE1.R)


oenv <- this.path:::.envvars(R_PROMPT   = "FILE1.R> ",
                             R_CONTINUE = "FILE1.R+ ")


## 'sys.path()' works with 'source()'
source(FILE1.R, verbose = FALSE)


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


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


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


## 'sys.path()' works with 'knitr::knit()'
if (requireNamespace("knitr")) {
    FILE2.Rmd <- tempfile(fileext = ".Rmd")
    FILE3.md <- tempfile(fileext = ".md")
    writeLines(c(
        "```{r}",
        ## same expression as above
        deparse(parse(FILE1.R)[[c(1L, 2L, 2L)]], width.cutoff = 55L),
        "```"
    ), FILE2.Rmd)
```

```
    # knitr::knit(FILE2.Rmd, output = FILE3.md, quiet = FALSE)
    ## the above does not work when using the 'Run examples' button in
    ## the HTML documentation. {knitr} cannot knit a document inside
    ## another document, pretty embarrassing oversight, so we have to
    ## launch a new R session and knit the document from there
    FILE4.R <- tempfile(fileext = ".R")
    this.path:::.write.code(bquote({
        knitr::knit(.(FILE2.Rmd), output = .(FILE3.md), quiet = TRUE)
    }), FILE4.R)
    this.path:::.Rscript(
        c("--default-packages=NULL", "--vanilla", FILE4.R),
        show.command = FALSE
    )
    unlink(FILE4.R)


    this.path:::.cat.file(FILE2.Rmd, number.nonblank = TRUE,
        squeeze.blank = TRUE, show.tabs = TRUE,
        show.command = TRUE)
    this.path:::.cat.file(FILE3.md, number.nonblank = TRUE,
        squeeze.blank = TRUE, show.tabs = TRUE,
        show.command = TRUE)
    unlink(c(FILE3.md, FILE2.Rmd))
}


## 'sys.path()' works with 'compiler::loadcmp()'
if (requireNamespace("compiler")) {
    FILE2.Rc <- tempfile(fileext = ".Rc")
    compiler::cmpfile(FILE1.R, FILE2.Rc)
    oenv2 <- this.path:::.envvars(R_PROMPT   = "FILE2.Rc> ",
                                  R_CONTINUE = "FILE2.Rc+ ")
    compiler::loadcmp(FILE2.Rc)
    this.path:::.envvars(oenv2)
    unlink(FILE2.Rc)
}


## 'sys.path()' works with 'box::use()'
if (requireNamespace("box")) {
    FILE2.R <- tempfile(fileext = ".R")
    this.path:::.write.code(bquote({
        ## we have to use box::set_script_path() because {box}
        ## does not allow us to import a module by its path
        script_path <- box::script_path()
        on.exit(box::set_script_path(script_path))
        box::set_script_path(.(normalizePath(FILE1.R, "/")))
        box::use(module = ./.(as.symbol(this.path::removeext(
            this.path::basename2(FILE1.R)
        ))))
        box::unload(module)
    }), FILE2.R)
    source(FILE2.R, echo = TRUE, spaced = FALSE, verbose = FALSE,
        prompt.echo = "FILE2.R> ", continue.echo = "FILE2.R+ ")
    unlink(FILE2.R)
}
```

```
## 'sys.path()' works with 'Rscript'
## it also works with other GUIs but that is
## not possible to show in a simple example
this.path:::.Rscript(c("--default-packages=NULL", "--vanilla", FILE1.R))
this.path:::.envvars(oenv)


## 'sys.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)


for (expr in c("sys.path()",
               "sys.path(default = NULL)",
               "sys.dir()",
               "sys.dir(default = NULL)",
               "sys.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::sys.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")
    })


unlink(FILE1.R)
```

`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)
```

---

this.path *Determine Script's Filename*

Description

`env.path()` returns the normalized path associated with the top level environment (see `?topenv`).

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

`src.path()` returns the normalized path associated with its source reference.

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

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

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

Usage

```
env.path(verbose = getOption("verbose"), original = FALSE,
         for.msg = FALSE, contents = FALSE, n = 0,
         envir = parent.frame(n + 1),
         matchThisEnv = getOption("topLevelEnvironment"),
         default, else.)
env.dir(verbose = getOption("verbose"), n = 0,
        envir = parent.frame(n + 1),
        matchThisEnv = getOption("topLevelEnvironment"),
        default, else.)

src.path(verbose = getOption("verbose"), original = FALSE,
         for.msg = FALSE, contents = FALSE, n = 0,
         srcfile = sys.call(if (n) sys.parent(n) else 0),
         default, else.)
src.dir(verbose = getOption("verbose"), n = 0,
        srcfile = sys.call(if (n) sys.parent(n) else 0),
        default, else.)

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 = sys.call(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 = sys.call(if (n) sys.parent(n) else 0),
         default, else.)
```

Arguments

`verbose, original, for.msg, contents, local, default, else.`	See ?`sys.path()`.
`n`	the number of additional generations to go back. By default, `this.path()` will look for a path based on the call to `this.path()` and the environment in which `this.path()` was called. This can be changed to be based on the call `n` generations up the call 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.

Details

There are two ways in which `env.path()` will find a path associated with the top level environment:

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

If `env.path()` does not find an associated path, it will throw an error.

`src.path()` will look for a source file in its argument. It will look at the bindings `filename` and `wd` to determine the associated file path. Filenames such as `""`, `"clipboard"`, and `"stdin"` will be ignored as 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 `src.path()` does not find an associated path, it will throw an error.

`this.path()` determines the path of the script in which it is written by:

1. examining its `srcfile` argument, looking for an associated path, the same as `src.path()`.
2. examining the top level environment, looking for an associated path, the same as `env.path()`.
3. examining the call stack, looking for the path of the executing script, the same as `sys.path()`.

`this.path()` and `this.dir()` are likely the functions you want to use. `env.path()`, `env.dir()`, `src.path()`, and `src.dir()` are provided for completeness / / convenience but are less general purpose. If you need to know the path of the executing script, perhaps for logging purposes, then you should use `sys.path()` and `sys.dir()`.

Value

character string.

Argument 'n'

By default, `this.path()` will look for a path based on 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.

Consider another scenario in which you 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.

Consider another scenario in which someone wishes to make a second function that uses `fun()`. They do not care where `fun()`, 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 are using `this.path()` without modularizing your code, i.e. relying on source references, you should be aware of how R stores source references. Only top level expressions will be given a source reference which includes expressions directly inside braces; sub-expressions will not receive a source reference. For example:

```
fun <- function ()
{
    ## this one will have a source reference
    ## because it is a top level expression
    x <- this.path::this.path(verbose = TRUE)
    print(x)
    ## this one will not have a source reference
    ## because it is just a sub-expression
    ## `print()` is the top level expression
    print(this.path::this.path(verbose = TRUE))
    ## this one will have a source reference
    ## note the surrounding braces
    print({ this.path::this.path(verbose = TRUE) })
    ## this one will have a source reference because `return()`
    ## and `list()` are not "closure" type functions, they are
    ## "special" and "builtin" type functions
    return(list(this.path::this.path(verbose = TRUE)))
}
```

One (somewhat extreme) attitude of defensive programming is to always wrap braces around your calls to `this.path()` as well as all related functions: `this.dir()`, `here()`, `ici()`, `this.proj()`, `rel2here()`, `rel2proj()`, `LINENO()`, `try.this.path()`, `path.functions()` (if `file` is missing), `check.path()`, `check.dir()`, and `check.proj()`, as well as the active bindings `FILE` and `LINE`.

See Also

`shFILE()`

Examples

```
## the important difference between 'this.path()' and 'sys.path()'


FILE1.R <- tempfile("FILE1-", fileext = ".R")
this.path:::.write.code({
```

```
    fun <- function() {
        cat("\n> this.path()\n")
        ## note the braces around 'this.path()'
        ## to ensure it is a top level expression
        print({ this.path::this.path(verbose = TRUE) })
        cat("\n> sys.path()\n")
        print({ this.path::sys.path(verbose = TRUE) })
    }
    ## 'this.path()' and 'sys.path()' should be identical because the
    ## executing script is the same as the script of the source file
    fun()
}, FILE1.R)
source(FILE1.R, verbose = FALSE, keep.source = TRUE)


FILE2.R <- tempfile("FILE2-", fileext = ".R")
this.path:::.write.code({
    ## 'this.path()' and 'sys.path()' should no longer be identical
    ## since FILE2.R is now the executing script, and FILE1.R is not
    fun()
}, FILE2.R)
source(FILE2.R, verbose = FALSE)


unlink(c(FILE1.R, FILE2.R))
```

`this.proj` *Construct Path to File, Starting with the Project's Directory*

Description

`sys.proj()`, `env.proj()`, `src.proj()`, and `this.proj()` construct paths to files starting with the project's root.

`reset.proj()` will reset the paths cached by these functions. This can be useful if you created a new project in your R session that you would like to be detected without the need to restart the R session.

Usage

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

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

reset.proj()
```

Arguments

`...`	further arguments passed to `path.join()`.
`local`	See `?sys.path()`.
`n`	See `?this.path()`.
`envir, matchThisEnv`	See `?env.path()`.
`srcfile`	See `?src.path()`.

Details

Unlike `here::here()`, these functions support sub-projects and multiple projects in use at once, and will choose which project root is appropriate based on `sys.dir()`, `env.dir()`, `src.dir()`, or `this.dir()`. Additionally, it is independent of working directory, whereas `here::here()` relies on the working directory being set within the project's directory when the package is loaded. Arguably, this makes it better than `here::here()`.

Value

A character vector of the arguments concatenated term-by-term, starting with the project's root.

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

Description

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

`try.env.path()`, `try.src.path()`, `try.this.path()`, and `try.shFILE` do the same with `env.path()`, `src.path()`, `this.path()`, and `shFILE()`.

Usage

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

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

try.shFILE()
```

Arguments

`contents, local`	See ?sys.path().
`n`	See ?this.path().
`envir, matchThisEnv`	See ?env.path().
`srcfile`	See ?src.path().

Details

This should **NOT** be used to construct file paths against the script's directory. This should exclusively be used in the scenario that you would like the normalized path of the executing script, perhaps for a diagnostic message, but it is not required to exist and can be a relative path or undefined.

Value

character string.

Examples

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

---

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 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) {
    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

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

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

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()
```

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?

`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.

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', `testthat::source_file()`, `knitr::knit()`, and `compiler::loadcmp()`. 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(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', `testthat::source_file()`, `knitr::knit()`, `compiler::loadcmp()`, and `box::use()` 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, ...)
        }
    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, ...)
        }
    knitr::knit() wrapper <- function(input, ...) {
            ## this works for the most part, but will not work in child mode
            input <- set.sys.path(input)
            knitr::knit(input = input, ...)
        }
    compiler::loadcmp() wrapper <- function(file, ...) {
            file <- set.sys.path(file, path.only = TRUE)
            compiler::loadcmp(file = file, ...)
        }
    box::use() ## 'box' is structured to be an incredible pain in the ass,
        ## so this solution is not exactly ideal, but it works so whatever
        wrapper <- function(path, ...) {
            p <- this.path::path.split.1(path.expand(path))
            n <- length(p)
            prefix <- if (n < 2L)
                "."
            else p[-n]
            name <- p[[n]]
            if (dir.exists(path)) {
                paths <- this.path::path.join(path, paste0("__init__",
                    c(".r", ".R")))
                paths <- paths[file.exists(paths)]
                if (!length(paths))
                    stop(sprintf(
                        "no file \"__init__.r\" or \"__init__.R\" found in %s",
                        encodeString(path, quote = "\"")))
                path <- paths[[1L]]
            }
            else {
                x <- this.path::splitext(name)
                if (x[[2L]] %in% c(".r", ".R"))
                    name <- x[[1L]]
            }
```

```
        spec <- list(name = name, prefix = prefix, attach = NULL,
            alias = name, explicit = FALSE)
        class(spec) <- c("box$mod_spec", "box$spec")
        path <- set.sys.path(path, path.only = TRUE)
        info <- list(name = name, source_path =
            if (.Platform$OS.type == "windows")
            chartr("/", "\\", path) else path)
        class(info) <- c("box$mod_info", "box$info")
        caller <- parent.frame()
        box:::load_and_register(spec, info, caller)
        invisible()
    }
```

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(n = -1, text = lines, srcfile = srcfile)
    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(n = -1, text = lines, srcfile = srcfile)
    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(n = -1, text = lines, srcfile = srcfile)
    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(n = -1, text = lines, srcfile = srcfile)
    source.exprs(exprs, evaluated = TRUE, envir = envir)
```

```
    envir
}


sourcelike6(FILE.R)


unlink(FILE.R)
```

Index