

# Package ‘this.path’

November 6, 2025

**Version** 2.7.1

**Date** 2025-11-07

**License** MIT + file LICENSE

**Title** Get Executing Script's Path

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

**Author** Iris Simmons [aut, cre]

**Maintainer** Iris Simmons <ikwsimmo@gmail.com>

**Depends** R (>= 2.15)

**Suggests** utils, jsonlite, microbenchmark, rprojroot

**Enhances** compiler, box, knitr, plumber, shiny, targets, testthat

**URL** <https://github.com/ArcadeAntics/this.path>

**BugReports** <https://github.com/ArcadeAntics/this.path/issues>

**ByteCompile** TRUE

**Biarch** TRUE

**BuildManual** TRUE

**Type** Package

## Contents

this.path-package	2
basename2	4
check.path	5
Enhances	6
ext	7
here	8
LINENO	10
make_fix_funs	11

OS.type . . . . .	12
path.join . . . . .	13
path.split . . . . .	13
relpath . . . . .	14
set.gui.path . . . . .	16
set.jupyter.path . . . . .	18
set.sys.path . . . . .	19
shFILE . . . . .	26
startup_files . . . . .	27
this.path . . . . .	28
try.this.path . . . . .	34

**Index****36**

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

**Description**

Determine the path of the executing script.

Compatible with several popular GUIs:

- ‘Rgui’
- ‘**RStudio**’ (including **background jobs**)
- ‘**Positron**’
- ‘**VSCode**’ + ‘**REditorSupport**’
- ‘**Jupyter**’
- ‘**Emacs**’ + ‘**ESS**’
- ‘**Rscript**’ (shell)

Compatible with several functions and packages:

- **source()**
- **sys.source()**
- **debugSource()** in ‘**RStudio**’
- **compiler::loadcmp()**
- **utils::Sweave()**
- **box::use()**
- **knitr::knit()**
- **plumber::plumb()**
- **shiny::runApp()**
- **package:targets**
- **testthat::source\_file()**

## Details

The most important functions from **package: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()`.

New additions include:

- `LINENO()` returns the line number of the executing expression.
- `shFILE()` looks through the command line arguments, extracting ‘FILE’ from either of the following: ‘-f’ ‘FILE’ or ‘--file=FILE’
- `set.sys.path()` implements `this.path()` for any `source()`-like functions outside of the builtins.
- `with_init.file()` allows `this.path()` and related to be used in a user profile.

**package: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.

## Note

This package started from a stack overflow posting:

<https://stackoverflow.com/questions/1815606/determine-path-of-the-executing-script/64129649#64129649>

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 `bug.report(package = "this.path")` to report your issue.

## Author(s)

Iris Simmons [aut, cre]

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, expand = TRUE)
dirname2(path, expand = TRUE)
```

**Arguments**

path	character vector, containing path names.
expand	logical. Should tilde (see <code>path.expand</code> ) be expanded?

**Details**

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 empty 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 empty 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 up to the project root.

## 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 'd_cead'
# ##
# ## 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("d_cead", "code", "provrun.R")
#
# ## or:
#
# this.path::check.path("d_cead/code/provrun.R")
```

## Description

These functions improve the user experience of other packages.

## Usage

```
## enhances 'package:box'
with_script_path(expr, file, local = FALSE, n = 0, envir = parent.frame(n + 1),
  matchThisEnv = getOption("topLevelEnvironment"),
  srcfile = if (n) sys.parent(n) else 0)

## enhances 'package:rprojroot'
make_fix_file(criterion, local = FALSE, n = 0, envir = parent.frame(n + 1),
  matchThisEnv = getOption("topLevelEnvironment"),
  srcfile = if (n) sys.parent(n) else 0)
```

## Arguments

expr	an expression to evaluate after setting the current script in <b>package:box</b> ; most commonly a call to <code>box::use()</code> .
file	a character string giving the pathname of the file.
criterion	argument passed to <code>rprojroot::find_root()</code> .
local, n, envir, matchThisEnv, srcfile	See <code>?this.path()</code> .

## Details

`with_script_path()` improves the experience of **package:box**; it sets the current script in **package:box** to file or `this.path()` using `box::set_script_path()`, then evaluates its argument, most commonly a **package:box** import statement.

`make_fix_file()` improves the experience of **package:rprojroot**; it looks for a project root starting with `this.dir()`, then makes a function that constructs file paths against said project root.

**Value**

for `with_script_path()`, the result of evaluating `expr`.  
 for `make_fix_file()`, a function with formals ( $\dots, \dots = 0$ ) that returns a character vector.

**Examples**

```
# this.path::with_script_path(
# box::use(
#   <import 1>,
#   <import 2>,
#   <...>
# )
# )

# ## replace 'rprojroot::is_r_package' with desired criterion
#
# fix_file <- this.path::make_fix_file(rprojroot::is_r_package)
```

**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, expand = TRUE)
removeext(path, compression = FALSE, expand = TRUE)
ext(path, compression = FALSE, expand = TRUE)
ext(path, compression = FALSE, expand = TRUE) <- value
```

**Arguments**

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

**Details**

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_2.7.1.tar.gz"
ext(path) <- ".png"
path

path <- "this.path_2.7.1.tar.gz"
ext(path, compression = TRUE) <- ".png"
path
```

here

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

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

**Usage**

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

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

reset.proj()

## alias for 'here'
ici(..., local = FALSE, n = 0,
```

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

## Arguments

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

## Details

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

## Value

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

## Examples

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

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

FILE.R <- this.path::path.join(tmpdir, "src", "R", "script1.R")
dir.create(this.path::dirname2(FILE.R), recursive = TRUE)
this.path::::writeCode({
  this.path::this.path()
  this.path::this.proj()
  ## use 'here' to run another script located nearby
  this.path::here("script2.R")
  ## or maybe to read input from a file
  this.path::here(.. = 2, "input", "data1.csv")
  ## but sometimes it is easier to use the project root
  ## this allows you to move the R script up or down
  ## a directory without changing the .. number
  this.path::this.proj("input", "data1.csv")
}, FILE.R)

source(FILE.R, echo = TRUE)

unlink(tmpdir, recursive = TRUE)
```

---

LINENO	<i>Line Number of Executing Expression</i>
--------	--

---

## Description

Get the line number of the executing expression.

## Usage

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

## Arguments

n, envir, matchThisEnv, srcfile  
 See [?this.path\(\)](#).

## Details

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

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

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

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

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

For `utils::Sweave()`, the `srcref` is never stored, there is nothing that can be done.

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

For `package:targets`, set option "keep.source" to `TRUE` before calling associated functions.

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

## Value

`integer; NA_integer_` if the line number is not found.

## Note

You can get a more accurate line number by wrapping `LINENO()` in braces:

```
{ LINENO() }
```

## 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)

if (getRversion() >= "4.3.0") {
  source(FILE.R, echo = TRUE, verbose = FALSE,
        max.deparse.length = Inf, keep.source = TRUE)
} else {
  this.path:::source(FILE.R, echo = TRUE, verbose = FALSE,
                    max.deparse.length = Inf, keep.source = TRUE)
}

unlink(FILE.R)

```

**make\_fix\_funs**

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

## Description

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

## Usage

```

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

## alias for 'make_fix_funs'
path.functions(file, delayed = FALSE, local = FALSE, n = 0,
               envir = parent.frame(n + 1),
               matchThisEnv = getOption("topLevelEnvironment"),
               srcfile = if (n) sys.parent(n) else 0)

```

## Arguments

<code>file</code>	a character string giving the pathname of the file or URL.
<code>delayed</code>	TRUE or FALSE; should the normalizing of <code>file</code> be delayed?
<code>local, n, envir, matchThisEnv, srcfile</code>	See <code>?this.path()</code> .

**Value**

An environment with at least the following bindings:

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

## 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](http://web.archive.org/web/20191012035921/http://nadeausoftware.com/articles/2012/01/c_c_tip_how_use_compiler_p)

---

path.join	<i>Construct Path to File</i>
-----------	-------------------------------

---

## 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	<i>Split File Path Into Individual Components</i>
------------	---

---

## 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))
```

**Description**

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

**Usage**

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

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

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

**Arguments**

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

**Details**

Tilde-expansion (see [?path.expand\(\)](#)) is first done on `path` and `relative.to`. If `path` and `relative.to` are equivalent, “.” will be returned. If `path` and `relative.to` have no base in common, the normalized path will be returned.

**Value**

character vector of the same length as `path`.

**Examples**

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

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

    "C:/Users/effective_user/Documents/testing",
    "C:\\Users\\\\effective_user",
    "C:/Users/effective_user/Documents/R/thispath.R"
  ),
  relative.to = "C:/Users/effective_user/Documents/this.path/man"
)
## End(Not run)
```

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

## Description

`set.gui.path()` can be used to implement `this.path()` for arbitrary GUIs.

## Usage

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

thisPathNotFoundError(..., call. = TRUE, domain = NULL,
                     call = .getCurrentCall())

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

## Arguments

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

## Details

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

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

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

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

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

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

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

```
evalq(envir = new.env(parent = .BaseNamespaceEnv), {
  .guiname <- "myGui"
  .custom_gui_path <- function(verbose, original, for.msg, contents) {
```

```

        if (verbose)
            cat("Source: document in", .guiname, "\n")

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

        ## if no documents are open, 'doc' should be NULL
        ## or some other object to represent no documents open
        if (is.null(doc)) {
            if (for.msg)
                NA_character_
            else stop(this.path::thisPathNotFoundError(
                "R is running from ", .guiname, " with no documents open\n",
                " (or document has no path)"))
        }
        else if (contents) {
            ## somehow, get and return the contents of the document
            <doc$contents>
        }
        else {
            ## somehow, get the path of the document
            path <- <doc$path>
            if (nzchar(path)) {
                ## if the path is not normalized, this will normalize it
                if (isFALSE(original))
                    normalizePath(path, "/", TRUE)
                else path
                # ## otherwise, you could just do:
                # path
            }
            else if (for.msg)
                ## return "Untitled" possibly translated
                gettext("Untitled", domain = "RGui", trim = FALSE)
            else
                stop(this.path::thisPathNotFoundError(
                    "document in ", .guiname, " has no associated path (has yet to be saved)"))
        }
    }
    ## recommended to prevent tampering
    lockEnvironment(environment(), bindings = TRUE)
    setHook(packageEvent("this.path"),
    function(pkgname, pkgpath) {
        this.path::set.gui.path(.custom_gui_path)
    }, action = "prepend")
})

```

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

```

evalq(envir = new.env(parent = .BaseNamespaceEnv), {
    .guiname <- "myGui"
    .path <- "~/example.R"
    .custom_get_contents <- function(path) {

```

```

## get the contents of the document
readLines(path, warn = FALSE)
}
## recommended to prevent tampering
lockEnvironment(environment(), bindings = TRUE)
setHook(packageEvent("this.path"), function(pkgname, pkgpath) {
  this.path::set.gui.path(.guiname, .path, .custom_get_contents)
}, action = "prepend")
# ## if your GUI does not have/need a .custom_get_contents
# ## function, then this works just as well:
# setHook(packageEvent("this.path"), function(pkgname, pkgpath) {
#   this.path::set.gui.path(.guiname, .path)
# }, action = "prepend")
})

```

### Value

a list of the previous settings for `set.gui.path()`, similar to `options()`.

`set.jupyter.path`

*Declare Executing 'Jupyter' Notebook's Filename*

### Description

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

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

### Usage

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

### Arguments

...	further arguments passed to <code>path.join()</code> . If no arguments are provided or exactly one argument is provided that is NA or NULL, the ‘Jupyter’ path is unset.
-----	--

### Details

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

### Value

character string, invisibly; the declared path for ‘Jupyter’.

## Examples

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

set.sys.path

*Implement 'this.path()' For Arbitrary 'source()'‑Like Functions*

## Description

`sys.path()` is implemented to work with these functions and packages:

- `source()`
- `sys.source()`
- `debugSource()` in ‘RStudio’
- `compiler::loadcmp()`
- `utils::Sweave()`
- `box::use()`
- `knitr::knit()`
- `plumber::plumb()`
- `shiny::runApp()`
- **package:targets**
- `testthat::source_file()`

`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 alongside `set.sys.path()` to implement `env.path()` and `src.path()`, thereby fully implementing `this.path()`. Note that `set.env.path()` only makes sense if the code is being modularized, see [Examples](#).

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

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

`with_sys.path()` is a convenient way to evaluate code within the context of a file. Whereas `set.sys.path()` can only be used within a function, `with_sys.path()` can only be used outside a function.

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.

`wrap.source()` should not be used, save for one specific use-case. See details.

**Usage**

```

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, delayed = FALSE)

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

set.src.path(srcfile)

unset.sys.path()

set.sys.path.function(fun)

with_sys.path(file, expr, ...)

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)

```

## Arguments

<code>expr</code>	for <code>with_sys.path()</code> , an expression to evaluate within the context of a file. for <code>wrap.source()</code> , an (unevaluated) call to a <code>source()</code> -like function.
<code>file</code>	a connection or a character string giving the pathname of the file or URL to read from.
<code>path.only</code>	must <code>file</code> be an existing path? This implies <code>character.only</code> and <code>file.only</code> are TRUE and implies <code>allow.file.uri</code> and <code>allow.unz</code> are FALSE, though these can be manually changed.
<code>character.only</code>	must <code>file</code> be a character string?
<code>file.only</code>	must <code>file</code> refer to an existing file?
<code>conv2utf8</code>	if <code>file</code> is a character string, should it be converted to UTF-8?
<code>allow.blank.string</code>	may <code>file</code> be a blank string, i.e. ""?
<code>allow.clipboard</code>	may <code>file</code> be "clipboard" or a clipboard connection?
<code>allow.stdin</code>	may <code>file</code> be "stdin"? Note that "stdin" refers to the C-level 'standard input' of the process, differing from <code>stdin()</code> which refers to the R-level 'standard input'.
<code>allow.url</code>	may <code>file</code> be a URL pathname or a connection of class "url-libcurl" // "url-wininet"?
<code>allow.file.uri</code>	may <code>file</code> be a 'file://' URL?
<code>allow.unz, allow.pipe, allow.terminal, allow.textConnection,</code>	
<code>allow.rawConnection, allow.sockconn, allow.servsockconn</code>	may <code>file</code> be a connection of class "unz" // "pipe" // "terminal" // "textConnection" // "rawConnection" // "sockconn" // "servsockconn"?
<code>allow.customConnection</code>	may <code>file</code> be a custom connection?
<code>ignore.all, ignore.blank.string, ignore.clipboard, ignore.stdin,</code>	ignore the special meaning of these types of strings, treating it as a path instead?
<code>ignore.url, ignore.file.uri</code>	
<code>Function</code>	character vector of length 1 or 2; the name of the function and package in which <code>set.sys.path()</code> is called.
<code>ofile</code>	a connection or a character string specifying the original file argument. This overwrites the value returned by <code>sys.path(original = TRUE)</code> .
<code>delayed</code>	TRUE or FALSE; should the normalizing of the path be delayed? Mostly for use with <code>make_fix_funs()</code> and similar.

```

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

srcfile      source file in which to assign a pathname.

fun          function with a srcref.

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

```

## 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 most **source()**-like functions. 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 **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)
}

for set.env.path(), envir invisibly.
for set.src.path(), srcfile invisibly.
for unset.sys.path() and set.sys.path.function(), NULL invisibly.
for with_sys.path() and wrap.source(), the result of evaluating expr.

```

### 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("'%" 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("'%" is not an existing file", ofile))
  file <- this.path::set.sys.path(file, ofile = ofile)
  exprs <- parse(n = -1, file = file)
  for (i in seq_along(exprs)) eval(exprs[i], envir)
  invisible()
}
```

### Examples

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

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

```

        }
        srcfile <- srcfilecopy(filename, lines, timestamp, isFile)
        set.src.path(srcfile)
        exprs <- parse(text = lines, srcfile = srcfile, keep.source = FALSE)
        invisible(source.exprs(exprs, evaluated = TRUE, envir = envir))
    }

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

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

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

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

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

## it is much easier to define a new function to do this
sourcelike3 <- function (...)

wrap.source(sourcelike2(...), path.only = TRUE)

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

## however, this is preferable:
sourcelike4 <- function (pathname, ...)

```

```

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

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

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

sourcelike5(FILE.R)

## the code can be made much simpler in some cases
sourcelike6 <- function (file)
{
  ## we expect a character string referring to a file
}
```

```

ofile <- file
filename <- set.sys.path(file, path.only = TRUE, ignore.all = TRUE,
    Function = "sourcelike6")
lines <- readLines(filename, warn = FALSE)
timestamp <- file.mtime(filename)[1]
srcfile <- srcfilecopy(filename, lines, timestamp, isFile = TRUE)
set.src.path(srcfile)
envir <- new.env(hash = TRUE, parent = .BaseNamespaceEnv)
envir$packageName <- filename
oopt <- options(topLevelEnvironment = envir)
on.exit(options(oopt))
set.env.path(envir)
exprs <- parse(text = lines, srcfile = srcfile, keep.source = FALSE)
source.exprs(exprs, evaluated = TRUE, envir = envir)
envir
}

sourcelike6(FILE.R)

unlink(FILE.R)

```

---

**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**

<code>original</code>	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.
<code>for.msg</code>	TRUE or FALSE; do you want the path for the purpose of printing a diagnostic message // warning // error? <code>for.msg = TRUE</code> will ignore <code>original = FALSE</code> , and will use <code>original = NA</code> instead.
<code>default</code>	if ‘FILE’ is not found, this value is returned.
<code>else.</code>	missing or a function to apply if ‘FILE’ is found. See <code>tryCatch2()</code> 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:::writeCode({
  this.path:::withAutoprint({
    this.path:::shFILE(original = TRUE)
    this.path:::shFILE()
    this.path:::shFILE(default = {
      stop("since 'FILE.R' will be found,\n",
           "this error will not be thrown")
    })
  }, spaced = TRUE, verbose = FALSE, width.cutoff = 60L)
}, FILE.R)
this.path:::Rscript(
  c("--default-packages=NULL", "--no-save", 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=NULL", "--no-save", "-e", expr)
  )
}
```

### Description

`site.file()` and `init.file()` return the normalized paths of the site-wide startup profile file and the user profile that were run at startup.

`with_init.file()` declares that the current script is the user profile then evaluates and auto-prints the sub-expressions of its argument.

**Usage**

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

with_site.file(expr)
with_init.file(expr)
```

**Arguments**

original, for.msg, default, else.	Same as <a href="#">shFILE()</a> .
expr	a braced expression, the sub-expressions of which to evaluate and auto-print.

**Value**

for `site.file()` and `init.file()`, a character string, or `default` if it was not found.  
 for `with_site.file()` and `with_init.file()`, `NULL` invisibly.

**Note**

`with_site.file()` is unneeded now that the site-wide startup profile file can be automatically detected.

**Examples**

```
## if you wish to use this.path() in a user profile,
## instead of writing:
##
## <expr 1>
## <expr 2>
## <...>
##
## write this:
##
## this.path::with_init.file({
## <expr 1>
## <expr 2>
## <...>
## })
```

---

`this.path`

*Determine Script's Filename*

---

**Description**

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

**Usage**

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

  

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

**Arguments**

verbose	TRUE or FALSE; should the method in which the path was determined be printed?
original	TRUE, FALSE, or NA; should the original or the normalized path be returned? NA means the normalized path will be returned if it has already been forced, and the original path otherwise.
for.msg	TRUE or FALSE; do you want the path for the purpose of printing a diagnostic message // warning // error? This will return NA_character_ in most cases where an error would have been thrown.  for.msg = TRUE will ignore original = FALSE, and will use original = NA instead.
contents	TRUE or FALSE; should the contents of the script be returned instead?  In 'Jupyter', a list of character vectors will be returned, the contents separated into cells. Otherwise, a character vector will be returned. If the executing script cannot be determined and for.msg is TRUE, NULL will be returned.  You could use as.character(unlist(this.path(contents = TRUE))) if you require a character vector.  This is intended for logging purposes. This is useful in 'Rgui', 'RStudio', 'VS-Code', and 'Emacs' when the source document has contents but no path.
local	TRUE or FALSE; should the search for the executing script be confined to the local environment in which <a href="#">set.sys.path()</a> was called?
n	the number of additional generations to go back. By default, this.path() will look for a path based on the srcref of the call to this.path() and the environment in which this.path() was called. This can be changed to be based on the srcref of the call and the calling environment n generations up the stack. See section <b>Argument 'n'</b> for more details.
envir, matchThisEnv	arguments passed to topenv() to determine the top level environment in which to search for an associated path.
srcfile	source file in which to search for a pathname, or an object containing a source file. This includes a source reference, a call, an expression object, or a closure.
default	this value is returned if there is no executing script.
else.	function to apply if there is an executing script. See <a href="#">tryCatch2()</a> for inspiration.

## Details

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

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

- from a **package:box** module's namespace
- from an attribute "path"

If it does not find an associated path with `envir` and `matchThisEnv`, it will next examine the call stack looking for a source call; a call to one of these functions:

- `source()`
- `sys.source()`
- **debugSource()** in '**RStudio**'
- `compiler::loadcmp()`
- `utils::Sweave()`
- `box::use()`
- `knitr::knit()`
- `plumber::plumb()`
- `shiny::runApp()`
- `targets::tar_callr_inner_try()`  
`targets::tar_load_globals()`  
`targets::tar_source()`  
`targets::tar_workspace()`
- `testthat::source_file()`

If a source call is found, the file argument is returned from the function's evaluation environment. If you have your own `source()`-like function that you would like to be recognized by `this.path()`, please use `set.sys.path()` or contact the package maintainer so that it can be implemented.

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

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

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

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

‘**RStudio**’ then the active document’s filename (the document in which the cursor is active) is returned (at the time of evaluation). If the active document is the R console, the source document’s filename (the document open in the current tab) is returned (at the time of evaluation). Please note that the source document will *NEVER* be a document open in another window (with the **Show in new window** button). Please also note that an active document open in another window can sometimes lose focus and become inactive, thus returning the incorrect path. It is best to not run R code from a document open in another window. It is important to not leave the current tab (either by closing or switching tabs) while any calls to `this.path()` have yet to be evaluated in the run selection. If no documents are open or the source document does not exist (not saved anywhere), an error is thrown.

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

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

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

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

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

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

‘**AQUA**’ then the executing script’s path cannot be determined. Until such a time that there is a method for requesting the path of an open document, consider using ‘RStudio’, ‘Positron’, ‘VSCode’, or ‘Emacs’.

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

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

## Value

`default` if there is no executing script.

If `contents` is `TRUE`, there are a variety of return values. If a custom GUI is implemented with `set.gui.path()`, any R object. If the executing script cannot be determined and `for.msg` is `TRUE`, then `NULL`. In **Jupyter**, a list of character vectors, the contents separated into cells. Otherwise, a character vector.

Otherwise, a character string.

### Argument 'n'

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

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

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

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

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

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

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

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

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

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

{
```

```

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

{

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

```

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

### Note

If you need to use `this.path()` inside a user profile, please use `with_init.file()`. i.e. instead of writing:

```

<expr 1>
<expr 2>
<...>

```

write this:

```

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

```

### See Also

[shFILE\(\)](#)  
[set.sys.path\(\)](#)

### Examples

```

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

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

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

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

## 'this.path()' works with 'testthat::source_file()'
if (requireNamespace("testthat"))

```

```

testthat::source_file(FILE1.R, chdir = FALSE, wrap = FALSE)

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

## 'this.path()' works with 'Rscript'
this.path:::Rscript(c(
  "--default-packages=NULL", "--no-save", FILE1.R
))

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

unlink(FILE1.R)

```

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

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

**Usage**

```

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

try.shFILE()

```

**Arguments**

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

**Details**

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

**Value**

character string.

**Examples**

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

# Index

\* package  
  this.path-package, 2

basename2, 3, 4

check.dir (check.path), 5

check.path, 5

check.proj (check.path), 5

dirname2, 3

dirname2 (basename2), 4

Enhances, 6

env.path, 19

ext, 3, 7

ext<- (ext), 7

here, 3, 8, 27

ici (here), 8

init.file (startup\_files), 27

LINENO, 3, 10

make\_fix\_file (Enhances), 6

make\_fix\_funs, 11, 21

OS.type, 12

path.functions (make\_fix\_funs), 11

path.join, 3, 5, 9, 13, 14, 18

path.split, 3, 13

path.split.1, 3

path.unsplit, 3

path.unsplit (path.split), 13

rel2here, 3

rel2here (relpath), 14

rel2proj, 3

rel2proj (relpath), 14

relpath, 3, 14

removeext, 3

removeext (ext), 7

reset.proj (here), 8

set.env.path (set.sys.path), 19

set.gui.path, 16, 31

set.jupyter.path, 18, 31

set.src.path (set.sys.path), 19

set.sys.path, 3, 19, 29, 30, 33

shFILE, 3, 26, 28, 33

site.file (startup\_files), 27

splitext, 3

splitext (ext), 7

src.path, 19

startup\_files, 27

sys.path, 19, 21

this.dir, 3, 6, 8

this.dir (this.path), 28

this.path, 3, 5, 6, 9–11, 15, 16, 18, 19, 27, 28, 34

this.path-package, 2

this.proj, 3

this.proj (here), 8

thisPathNotFoundError (set.gui.path), 16

thisPathNotFoundError (set.gui.path), 16

try.shFILE (try.this.path), 34

try.this.path, 34

tryCatch2, 29

unset.sys.path (set.sys.path), 19

with\_init.file, 3, 33

with\_init.file (startup\_files), 27

with\_script\_path (Enhances), 6

with\_site.file (startup\_files), 27

with\_sys.path (set.sys.path), 19

wrap.source (set.sys.path), 19