

# Package ‘essentials’

January 12, 2022

**Version** 0.2.0

**License** MIT + file LICENSE

**Title** Essential Functions not Included in Base R

**Description** Functions for converting objects to scalars (vectors of length 1)  
and a more inclusive definition of data that can be interpreted as numbers  
(numeric and complex alike).

**Author** Andrew Simmons

**Maintainer** Andrew Simmons <akwsimmo@gmail.com>

**Imports** graphics, grDevices, methods, utils, this.path

**Enhances** tcltk

**URL** <https://github.com/ArcadeAntics/essentials>

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

**Encoding** UTF-8

**NeedsCompilation** yes

**Archs** i386, x64

## R topics documented:

|                               |    |
|-------------------------------|----|
| essentials-package . . . . .  | 2  |
| .plapply . . . . .            | 3  |
| add.argument . . . . .        | 5  |
| add.help . . . . .            | 6  |
| add.subparsers . . . . .      | 7  |
| Args . . . . .                | 8  |
| ArgumentParser . . . . .      | 9  |
| as.colorRampPalette . . . . . | 11 |
| as.scalar . . . . .           | 12 |
| ASCII . . . . .               | 13 |
| aslength1 . . . . .           | 14 |
| asWindowsbasename . . . . .   | 15 |
| color.with.alpha . . . . .    | 16 |
| Commands . . . . .            | 17 |
| dedent . . . . .              | 18 |
| delayedAssign2 . . . . .      | 19 |
| do.while . . . . .            | 20 |

|                          |           |
|--------------------------|-----------|
| envvars                  | 22        |
| file.open                | 24        |
| flat.list                | 24        |
| GeneralizedExtremeValue  | 25        |
| hcl.colors2              | 27        |
| hypot                    | 28        |
| InverseDistanceWeighting | 30        |
| legend.dimensions        | 31        |
| list.files2              | 33        |
| listify                  | 34        |
| Missing                  | 35        |
| normalizeAgainst         | 36        |
| numbers                  | 36        |
| numbers-class            | 38        |
| parse.args               | 38        |
| path.contract            | 39        |
| plapply                  | 40        |
| progressBar              | 42        |
| pseudoglobalenv          | 43        |
| python                   | 43        |
| R                        | 44        |
| readArgs                 | 45        |
| rowmatch                 | 50        |
| Runge-Kutta Methods      | 51        |
| setReadWriteArgsMethod   | 55        |
| shEncode                 | 57        |
| shPrompt                 | 59        |
| strip                    | 60        |
| toProv                   | 61        |
| tryExcept                | 62        |
| withArgs                 | 65        |
| wrapper                  | 67        |
| <b>Index</b>             | <b>68</b> |

---

|                    |   |
|--------------------|---|
| essentials-package | <i>Essential Functions not Included in Base R</i> |
|--------------------|---|

---

## Description

Functions for converting objects to scalars (vectors of length 1) and a more inclusive definition of data that can be interpreted as numbers (numeric and complex alike).

## Details

The four most important functions from this package are `as.numbers`, `is.numbers`, `as.scalar` and `as.length1`.

`as.numbers` coerces its argument to type double or complex. `is.numbers` tests if its argument is interpretable as numbers.

`as.scalar` coerces its argument to an scalar (an atomic vector of length 1). It strips attributes including names.

`aslength1` coerces its argument to a vector of length 1 (not necessarily atomic). It strips attributes from arguments that are not vectors, but preserves names for arguments that are vectors.

### Author(s)

Andrew Simmons

Maintainer: Andrew Simmons <akwsimmo@gmail.com>

### Examples

```
as.numbers("4")
as.numbers("4+0i") # imaginary component is removed
as.numbers("4+1i")

is.numbers(4L)
is.numbers(4)
is.numbers(4+1i)

as.scalar(1:100)
as.scalar(as.list(1:100)) # coerced to NA_character_ since argument isn't atomic

aslength1(1:100) # identical to as.scalar(1:100)
aslength1(as.list(1:100)) # returns a list of length 1
```

---

*.plapply*

*Apply a Function to Multiple List or Vector Arguments*

---

### Description

Alternate versions of `plapply`, `psapply`, and `pvapply` which accept an argument `dots` instead of `...`

### Usage

```
.plapply(X, FUN, dots = NULL)

.psapply(X, FUN, dots = NULL, simplify = TRUE, USE.NAMES = TRUE)

.pvapply(X, FUN, FUN.VALUE, dots = NULL, USE.NAMES = TRUE)
```

### Arguments

|                        |   |
|------------------------|---|
| <code>X</code>         | list of arguments to vectorize over.  |
| <code>FUN</code>       | function to apply, found via <a href="#">match.fun</a> .  |
| <code>dots</code>      | list of optional arguments to <code>FUN</code> .  |
| <code>simplify</code>  | logical or character string; should the result be simplified to a vector, matrix, or higher dimensional array if possible? The default value, <code>TRUE</code> , returns a vector or matrix if appropriate, whereas if <code>simplify = "array"</code> the result may be an <a href="#">array</a> of higher dimension. |
| <code>USE.NAMES</code> | logical; if <code>TRUE</code> and one of <code>X</code> is character and of equal length to the result, use that element of <code>X</code> as <a href="#">names</a> for the result unless it had names already.   |
| <code>FUN.VALUE</code> | a (generalized) vector; a template for the return value from <code>FUN</code> .   |

## Details

While plapply would build a call like:

```
FUN(X[[1L]][[j]],X[[2L]][[j]],...)
```

.plapply builds a call like:

```
FUN(X[[1L]][[j]],X[[2L]][[j]],dots[[1L]],dots[[2L]])
```

with more or less dots[[k]] depending on the length of dots. Additionally, the dots arguments in the call will be named if dots has names.

dots is not coerced by base: `as.list`, instead its subsetting (`[[`), `length`, and `names` methods will be used.

## Note

plapply will only evaluate its ... argument as necessary, and even when forced, it will only evaluate the desired elements.

In contrast, .plapply will *ALWAYS* evaluate its dots argument, even if it is never used. This is because we need to know dots length and names to build the call.

## See Also

[plapply](#)

## Examples

```
# you should see here that plapply will not evaluate its
# optional arguments to FUN (because they are not used in
# this example)
#
# but .plapply will evaluate its optional arguments, even
# though they are not used in this example
invisible(essentials:: plapply(NA, function(...) {
  print(substitute(list(...)))
}), k = cat("evaluated optional arguments to FUN\n"))
invisible(essentials::.plapply(NA, function(...) {
  print(substitute(list(...)))
}), list(k = cat("evaluated optional arguments to FUN\n"))))

# also, plapply will only evaluate optional arguments as
# requested
invisible(essentials:: plapply(NA, function(x, ...) ..1,
  cat("evaluated first optional argument to FUN\n"),
  cat("evaluated second optional argument to FUN\n"))
invisible(essentials::.plapply(NA, function(x, ...) ..1,
  list(cat("evaluated first optional argument to FUN\n"),
  cat("evaluated second optional argument to FUN\n"))))
```

add.argument

*Declare a Formal Argument for an Argument Parser***Description**

Set the formal arguments of an [ArgumentParser](#).

**Usage**

```
## S4 method for signature 'essentials_ArgumentParser'
add.argument(..., action = NULL, nargs = NULL, constant, default,
  type = "any", choices = NULL, required = NA, help = NA,
  metavariable = NA, destination = NA, wrap = TRUE,
  overwrite = getOption("essentials.overwrite", TRUE))
```

**Arguments**

|              |   |
|--------------|---|
| ...          | character strings. A name for a positional argument or a set of flags for an optional argument.   |
| action       | character string. One of "store", "store_const", "store_true", "store_false", "append", "count", "help", "exit", and "skip", indicating what to do when this argument is encountered.   |
| nargs        | number of arguments expected for this formal argument. A positive number specifying exactly the accepted number of arguments, two numbers specifying the range of the accepted number of arguments, or a character string indicating the accepted number of arguments, see <b>Details</b> . |
| constant     | for action "store_const", the object to be stored when this argument is encountered.  |
| default      | the object to be stored when this argument is not encountered.  |
| type         | character string naming an atomic mode or "list" or "expression" or "any". If "any", the appropriate type is decided based on action and default.   |
| choices      |   |
| required     | logical. Is this argument required?   |
| help         | character vector. A brief description of the formal argument.   |
| metavariable | character string. A name for the value in usage messages.   |
| destination  | character string. A name for the assigned value returned by <a href="#">parse.args</a> .  |
| wrap         | logical. Should the usage message help be <a href="#">wrapped</a> when printing the help message for the argument parser?   |
| overwrite    | logical. If the name or flags are already in use, should they be replaced by this new definition?   |

**Details**

A positional argument is an argument in which the name of argument is not specified before its value (its value is assigned to the next available positional argument). The other types of arguments are arguments specified by a leading flag, either a short flag or a long flag.

action is an instruction for what to do when this argument is found at the command-line.

action = "store" Stores the argument's value. When multiple values are provided, the last value is stored and the others are discarded.

action = "store\_const" If this argument is provided, constant is stored, otherwise default is stored.

action = "store\_true", "store\_false" Special cases of action = "store\_const" in which constant = TRUE, default = FALSE and constant = FALSE, default = TRUE, respectively. default can be overwritten for special cases.

action = "append" Stores the argument's value. When multiple values are provided, the values are appended and stored together.

action = "count" Store the number of times this argument was provided.

action = "help" Print the help message and exit the program.

action = "exit" Print an exit message and exit the program.

action = "skip" Skip the rest of the command line (the following arguments are for another program).

nargs can be specified in three manners:

- A positive number, the exact number of arguments that must be provided. Specifying nargs in this manner does not make sense with action "store\_const", "store\_true", "store\_false", and "count", or with action "store" when nargs is greater than 1, and will signal a warning.
- Two non-negative numbers, a lower and upper bound on the number of arguments that must be provided. Specifying 0 as the lower bound indicates no lower limit, and Inf as the upper bound indicates no upper limit.
- A character string, specifically one of the [regex](#) repetition quantifiers.
  - ? At most one argument.
  - \* Zero or more arguments.
  - + One or more arguments.
  - {n} Exactly n arguments.
  - {n,} At least n arguments.
  - {n,m} At least n arguments, at most m arguments.

When nargs is NULL, required is used in its place.

## Value

Invisible NULL.

---

add.help

*Add Common Arguments to an ArgumentParser*

---

## Description

Convenience functions for adding help, skip, and version arguments to an ArgumentParser.

A help argument will print the help message when requested, a skip argument will indicate that the following arguments are for another program (and hence, will not be parsed by the parser), and a version argument will print the version of the running program.

**Usage**

```
## S4 method for signature 'essentials_ArgumentParser'
add.help(name.or.flags = c("-h", "--help"), action = "help",
         help = default.help("help"), wrap = FALSE, ...)

## S4 method for signature 'essentials_ArgumentParser'
add.skip(name.or.flags = "--args", action = "skip",
         help = default.help("skip"), wrap = FALSE, ...)

## S4 method for signature 'essentials_ArgumentParser'
add.version(name.or.flags = "--version", action = "exit",
            help = default.help("version"), wrap = FALSE, exit, ...)
```

**Arguments**

|               |   |
|---------------|---|
| name.or.flags | character vector. The name or flags of the argument.  |
| action        | character string. Indicates what to do when this argument is encountered.   |
| help          | character vector. A brief description of the formal argument.   |
| wrap          | logical. Should the usage message help be <a href="#">wrapped</a> when printing the help message for the argument parser? |
| exit          | character vector. The text to be printed before terminating the program.  |
| ...           | further arguments passed to <a href="#">add.argument</a> .  |

**Value**

Invisible NULL.

**Note**

Unlike Python's `argparse`, a version argument is a specific case of an exit argument, an argument which will print an exit message before the program is terminated. For example, on Unix, the `'RHOME'` argument for the `'R'` executable is an exit argument, and its exit message is the user's home directory.

---

|                |  |
|----------------|--|
| add.subparsers | <i>Declare a List of Sub-Commands for an <a href="#">ArgumentParser</a>.</i> |
|----------------|--|

---

**Description**

Split functionality of a script into any number of sub-commands.

**Usage**

```
## S4 method for signature 'essentials_ArgumentParser'
add.subparsers(title = NA, description = NA, program = NA, required = FALSE,
              wrap = TRUE, indent = 0, exdent = 0)
```

Arguments

|             |  |
|-------------|--|
| title       | character string. The name of the sub-commands group in the help message, by default "Commands:".  |
| description | character vector. A brief description of the sub-commands group.   |
| program     | character string. The name of the program to be displayed in the help message, by default the name of the original program followed by all previous sub-commands.  |
| required    | logical. Are one of these sub-commands required? For example, in R, the sub-command CMD is not required, whereas in R CMD, the sub-commands INSTALL, REMOVE, ... are required, you must provide one of them. |
| wrap        | logical. Should description be <b>wrapped</b> when printing the help message for the argument parser?  |
| indent      | a non-negative integer giving the indentation of the first line in a paragraph.  |
| exdent      | a non-negative integer giving the indentation of the subsequent lines in a paragraph.  |

Details

add.subparsers will return an object of class "subparsers", which can be used to add individual sub-commands with add.parser.

Value

An object of class "subparsers".

---

|      |   |
|------|---|
| Args | <i>Extract Arguments Supplied to a Script // From an Object of Class "ParsedArgs"</i> |
|------|---|

---

Description

Provides access to a copy of the arguments supplied to the current R script  
*OR*  
to the arguments supplied to an object of class "ParsedArgs".

Usage

Args(x, type = c("original", "all", "trailingOnly"))

Arguments

|      |   |
|------|---|
| x    | missing or an object of class "ParsedArgs".                         |
| type | missing or a string naming the type of arguments to extract from x. |



## Details

`Args()` is a replacement for `commandArgs()`. In addition to working from the command-line / terminal, it also works in combination with `withArgs`. When running from the command-line, only the arguments following `--args` will be returned. If not being run from the command-line or `withArgs`, `character(0)` is returned instead.

If `x` and `type` are provided, then `Args` extracts the arguments provided to that object of class "ParsedArgs". The types have the following meaning:

**"original"** original arguments supplied when `x` was created

**"all"** arguments after resolving all from-file arguments

**"trailingOnly"** arguments supplied after the "skip" argument

## Value

A character vector containing the user-supplied script arguments

*OR*

a character vector containing the arguments supplied when `x` was created.

## Examples

```
essentials:::write.code(file = FILE <- tempfile(), withAutoprint({

  # the regular command-line arguments
  commandArgs()

  # the script arguments
  essentials::Args()

}, verbose = FALSE))

essentials::Rscript("--default-packages=NULL", FILE, args = pi)
essentials::withArgs(source(FILE, verbose = FALSE), pi)

unlink(FILE)
```

## Description

Create an object of class "essentials\_ArgumentParser", for parsing the command-line arguments into a more usable form.

## Usage

```
ArgumentParser(program = NA, usage = NA, description = NA, epilogue = NA,
  add.help = TRUE,
  wrap = TRUE, indent = 0, exdent = 0,
  wrap.description = wrap,
    indent.description = indent,
    exdent.description = exdent,
  wrap.epilogue = wrap,
    indent.epilogue = indent,
    exdent.epilogue = exdent,
  style = NA, wrap.help = FALSE, help.help = default.help("help"),
  ..., help.message = NULL)
```

## Arguments

|   |  |
|---|--|
| program                                     | character string. The name of the program (by default, the <a href="#">basename</a> of the executing script).  |
| usage                                       | character string. The usage of the program (by default, determined by program and any formal arguments provided by <a href="#">add.argument</a> ).   |
| description, epilogue                       | character vector. Messages to display before and after the arguments help.   |
| add.help                                    | logical. Should a help option be added to the parser?  |
| wrap.help, help.help, ...                   | further arguments passed to <a href="#">add.help</a> .   |
| wrap, wrap.description, wrap.epilogue       | logical. Should the messages be <a href="#">wrapped</a> when printing the help message for the argument parser?  |
| indent, indent.description, indent.epilogue | a non-negative integer giving the indentation of the first line in a paragraph.  |
| exdent, exdent.description, exdent.epilogue | a non-negative integer giving the indentation of the subsequent lines in a paragraph.  |
| style                                       |  |
| help.message                                | character vector or NULL. The help message to display. If NULL, the help message will be created from usage, description, epilogue, and any arguments provided by <a href="#">add.argument</a> . |

## Details

All of the above arguments help form the documentation for your argument parser. The documentation will be printed in a similar manner to R's command-line documentation. First, the usage will be printed. This includes the name of the program (the [basename](#) of the executing script), the names of the formal arguments, and the names of the sub-commands (if any). Second, the description of the program will be printed

An object of class "essentials\_ArgumentParser" is a [reference](#) class. This allows us to use argument parsers in an almost identical manner to the [argparse](#) library from Python. All arguments to ArgumentParser are used for printing a help message.

## Value

An object of class "essentials\_ArgumentParser".

## Methods

`add.argument` Declare a formal argument for an argument parser.

`add.subparsers` Declare

## References

The `argparse` library for Python.

## Examples

```
x <- essentials::ArgumentParser()
x$print.help()

y <- essentials::ArgumentParser(description = "A description for the program",
  epilogue = c(" --- Final Message --- ",
    "A final message for the program, do not wrap this message"),
  wrap.epilogue = FALSE)
y$print.help()
```

---

|                     |                            |
|---------------------|----------------------------|
| as.colorRampPalette | <i>Color Interpolation</i> |
|---------------------|----------------------------|

---

## Description

Create a function that interpolates a set of colors to produce new color palettes.

## Usage

```
as.colorRampPalette(...)
```

## Arguments

... arguments to pass to `colorRamp`.

## Details

This is a preferable alternative to `colorRampPalette` since the returned function has more arguments. The returned function has arguments `start`, `end`, `alpha`, and `rev`, similar to `rainbow` or `viridis.colors`.

If only one argument is provided that happens to be a function, it is assumed that this function is the function that maps the interval  $[0,1]$  to a series of colors, and so the argument will *not* be passed to `colorRamp`.

## Value

A function which takes an integer `n` and returns a character vector of `n` colors.

**Examples**

```
crp <- as.colorRampPalette(1:9)

show.colors(crp(20))

show.colors(crp(20, end = 19/20))

show.colors(crp(20, alpha = 0.75))

show.colors(crp(20, rev = TRUE))

show.colors(crp(20, end = 19/20, alpha = 0.75, rev = TRUE))
```

---

|           |                |
|-----------|----------------|
| as.scalar | <i>Scalars</i> |
|-----------|----------------|

---

**Description**

Coerce objects to scalars (vectors of length 1).

**Usage**

```
as.scalar(x, mode = "any")

as.scalar.logical(x)

as.scalar.integer(x)

as.scalar.real(x)
as.scalar.double(x)
as.scalar.numeric(x)

as.scalar.complex(x)

as.scalar.string(x)
as.scalar.character(x)

as.scalar.raw(x)

as.scalar.number(x, strict = TRUE)
```

**Arguments**

|        |  |
|--------|--|
| x      | object to be coerced.  |
| mode   | character string naming an atomic mode or "any".   |
| strict | TRUE or FALSE, should a complex number that is strictly real (real component is NA or NaN or imaginary component is NA or NaN or 0) be converted to a real number? |

**Details**

`as.scalar.logical` coerces an object to a vector of type “logical” of length 1.

`as.scalar.integer` coerces an object to a vector of type “integer” of length 1.

`as.scalar.real`, `as.scalar.double` and `as.scalar.numeric` coerces an object to a vector of type “numeric” of length 1.

`as.scalar.complex` coerces an object to a vector of type “complex” of length 1.

`as.scalar.number` coerces an object to a vector of type “numeric” or “complex” of length 1.

`as.scalar.string` and `as.scalar.character` coerces an object to a vector of type “character” of length 1.

`as.scalar` coerces an object to a vector of length 1 of a specified mode.

**Value**

A vector of length 1.

**Examples**

```
## if the type converting from and converting to are identical, as.scalar is a
## much shorter way of writing what you intend.
as.scalar(c(TRUE, FALSE, NA))
as.scalar(1:100)
as.scalar(1:10 + 0.5)
as.scalar(exp((0+1i) * 6 * (-4:4)))
as.scalar(letters)

## if the type converting from and converting to are not identical, it is better
## to specify the type converting to.
as.scalar.logical(c(TRUE, FALSE, NA))
as.scalar.integer(c(TRUE, FALSE, NA))
as.scalar.numeric(c(TRUE, FALSE, NA))
as.scalar.complex(c(TRUE, FALSE, NA))
as.scalar.character(c(TRUE, FALSE, NA))

as.scalar(TRUE, "character")
```

---

ASCII

---

*ASCII Characters*


---

**Description**

ASCII generates a character vector of the ASCII characters. If `plot = TRUE`, the resulting characters are plotted, before they are returned invisibly.

**Usage**

```
ASCII(extended = TRUE, cex = par("cex"), family = par("family"),
      mar = c(0, 2.1, 2.1, 0), plot = TRUE, warn.unused = TRUE)
```

**Arguments**

|             |   |
|-------------|---|
| extended    | logical. Should the extended ASCII character set be returned? Note that “Extended ASCII” does <i>NOT</i> mean that the ASCII standard has been updated to include more than 128 characters. |
| cex         | A numerical value giving the amount by which text should be magnified relative to the default.  |
| family      | The name of a font family for drawing text. See <a href="#">par</a> .   |
| mar         | A numerical vector of the form <code>c(bottom, left, top, right)</code> which gives the number of lines of margin to be specified on the four sides of the plot.                            |
| plot        | logical. If TRUE (default), the ASCII characters are plotted, and the ASCII characters are returned invisibly. Otherwise, the ASCII characters are returned.                                |
| warn.unused | logical. If <code>plot = FALSE</code> and <code>warn.unused = TRUE</code> , a warning will be issued when graphical parameters are passed to ASCII.   |

**Details**

The first 32 ASCII characters are control characters (as well as the 128-th, that is “\x7F” or “\177”), consisting of non-printable characters and whitespace characters, so they will likely plot as an empty box or nothing at all.

Characters “\x81”, “\x8D”, “\x8F”, “\x90”, and “\x9D” (or in octal notation as “\201”, “\215”, “\217”, “\220”, and “\235”) are unused in extended ASCII, and will plot unusually. In an ISO8859-1 locale, they seem to plot empty. In a UTF-8 locale, they seem to plot as their bytes codes (that is <81>, <8D>, <8F>, <90>, and <9D>). In other locales, they may plot as question marks, or any other unusual behaviour.

Characters “\x20” and “\xA0” are space and non-breaking space, so they will plot empty.

Character “\xAD” is a soft hyphen. In an ISO8859-1 locale, it appears the same as a regular hyphen (“\x2D”). In a UTF-8 locale, it appears empty. Unknown behaviour for other locales. This character may also plot differently depending on family.

**Value**

character vector, the ASCII character set. When extended, 255 characters, otherwise 127 characters (NUL character is not included since R does not allow nul character within strings).

if plot, returned invisibly.

**Examples**

```
ASCII()
```

---

aslength1

---

*Subset the First Element of a Vector*


---

**Description**

Subset the first element of a vector.

**Usage**

```
aslength1(x)
```

**Arguments**

`x` vector (or an object which can be coerced) with at least one element.

**Details**

Vectors of length one return themselves. Vectors of length greater than one return the first element with a warning. Vectors of length zero raise an error. If `x` is a vector (determined by `is.vector()`), names will be preserved.

**Value**

A vector of length 1.

**Examples**

```
aslength1(1)
aslength1(1:10)
try(aslength1(integer(0)))

print(system.file("R", "aslength1.Rd", package = "essentials"))
```

---

|                   |   |
|-------------------|---|
| asWindowsbasename | <i>Create a Basename Valid in Windows</i> |
|-------------------|---|

---

**Description**

Windows basenames must not contain the control characters "`\001`" through "`\037`", double quote `"`, asterisk `*`, slash `/`, colon `:`, left and right chevrons `<>`, question mark `?`, backslash `\`, and vertical bar `|`. Additionally, a Windows basename must not begin with a space, and must not end with a space or full stop. A basename which is valid in Windows is likely to be valid on any OS.

**Usage**

```
asWindowsbasename(path)
```

**Arguments**

`path` character vector; the strings to make into valid paths.

**Value**

A character vector, the same length and attributes as `path` (after possible coercion to character).

**Examples**

```
asWindowsbasename(c(
  " test ",
  " testing?.",
  "already valid name"
))
```

---

|                  |                      |
|------------------|----------------------|
| color.with.alpha | <i>Color Opacity</i> |
|------------------|----------------------|

---

**Description**

Change the alpha (opacity) of a series of colors.

**Usage**

```
color.with.alpha(x, alpha)
as.hex.code(x)
```

**Arguments**

|       |   |
|-------|---|
| x     | a character vector, a matrix as produced by <code>col2rgb</code> , or a matrix or data.frame where the first three columns are the red, green, and blue values. |
| alpha | an alpha-transparency level in the range [0,1] (0 means transparent and 1 means opaque).  |

**Details**

When alpha is missing, the alpha-transparency is taken from x. In that case, `color.with.alpha` is just decoding the colors into a character vector of hex colors codes, and not actually changing the opacity of the colors.

**Value**

A character vector of hex color codes.

**Examples**

```
example.colors <- c("red", "#FFA50099", "yellow", "green", "blue", "purple")
show.colors(example.colors)
show.colors(color.with.alpha(example.colors, alpha = NULL)) # remove all opacity
show.colors(color.with.alpha(example.colors, alpha = 0.25)) # one quarter opaque
show.colors(color.with.alpha(example.colors, alpha = 0.5))  # half opaque
show.colors(color.with.alpha(example.colors, alpha = 0.75)) # three quarters opaque
show.colors(color.with.alpha(example.colors, alpha = 1))    # fully opaque, same as NULL
```

```
example.colors2 <- 1:8
show.colors(example.colors2)
show.colors(color.with.alpha(example.colors2, alpha = 0.5))
show.colors(color.with.alpha(example.colors2, alpha = 0.75))
```

```
example.colors3 <- grDevices::col2rgb(gg.colors(10))
```

```
## show.colors would not be able to show example.colors3 because it's a matrix
## use as.hex.codes to decode the values to their hex codes, then display
show.colors(as.hex.code(example.colors3))
show.colors(color.with.alpha(example.colors3, alpha = 0.5))
```



```
show.colors(color.with.alpha(example.colors3, alpha = 0.75))

example.colors4 <- data.frame(
  red   = c( 75, 66, 24,  0,  0,  0,  0, 108, 187, 253)/255,
  green = c(  0, 44, 80, 112, 142, 168, 190, 208, 221, 227)/255,
  blue  = c( 85, 112, 134, 148, 152, 144, 125, 94, 56, 51)/255
)
show.colors(as.hex.code(example.colors4))
show.colors(color.with.alpha(example.colors4, alpha = 0.5))
show.colors(color.with.alpha(example.colors4, alpha = 0.75))

# alpha can be a vector, here we provide 10 different alpha values
show.colors(color.with.alpha(example.colors4,
  alpha = seq.int(0.5, 1, length.out = 10)))
```

---

Commands

---

*Extract Commands From an Object of Class "ParsedArgs"*


---

## Description

Provides access to a copy of the commands supplied to an object of class "ParsedArgs".

## Usage

```
Commands(x, type = c("original", "string"))
```

## Arguments

|      |                                     |
|------|-------------------------------------|
| x    | an object of class "ParsedArgs".    |
| type | a string naming the type to return. |

## Value

If type = "original", a character vector of the commands supplied when x was created.

If type = "string", a character string of the commands supplied when x was created, separated by "/".

## Examples

```
parser <- essentials::ArgumentParser()
`parser CMD1` <- parser$add.parser("CMD1")
`parser CMD2` <- parser$add.parser("CMD2")
`parser CMD1 a` <- `parser CMD1`$add.parser(c("a", "b"))
`parser CMD1 c` <- `parser CMD1`$add.parser(c("c", "d"))
`parser CMD2 e` <- `parser CMD2`$add.parser(c("e", "f"))
`parser CMD2 g` <- `parser CMD2`$add.parser(c("g", "h"))

essentials::Commands(parser$parse.args())
essentials::Commands(parser$parse.args(c("CMD1")))
essentials::Commands(parser$parse.args(c("CMD2")))
essentials::Commands(parser$parse.args(c("CMD1", "a")))
```

```

essentials::Commands(parser$parse.args(c("CMD1", "b")))
essentials::Commands(parser$parse.args(c("CMD1", "c")))
essentials::Commands(parser$parse.args(c("CMD1", "d")))
essentials::Commands(parser$parse.args(c("CMD2", "e")))
essentials::Commands(parser$parse.args(c("CMD2", "f")))
essentials::Commands(parser$parse.args(c("CMD2", "g")))
essentials::Commands(parser$parse.args(c("CMD2", "h")))

```

---

dedent

---

*Remove Common Leading Whitespace*


---

## Description

Remove the common leading whitespace from each line. This is useful in multi-line strings to make them line up with the left edge of the display, while still presenting them in indented form within the source code.

## Usage

```
dedent(x, strip = TRUE)
```

## Arguments

|       |  |
|-------|--|
| x     | character vector. The strings from which to remove common leading whitespace.  |
| strip | logical. Should the leading and trailing whitespace line be removed from each element of x? This will not remove the leading indent. |

## Details

strip is TRUE by default because the standard use case is to dedent a multi-line string that appears in the source code. Refer to section **Examples**.

Tabs and spaces are both whitespace, but they are not treated equally. This is because tab takes up a different number of spaces depending on the context.

## Value

character vector, the same length as x, and with the dim, dimnames, and names attributes of x (after possible coercion to character).

## Examples

```

cat(dedent("
  here is a multi-line string that appears in the source code. we wish to
  remove the common indent from each line, and dedent should do this for us!
  ---- hopefully this works ----
"), sep = "\n")

# for me, a tab prints as eight spaces when preceded by a newline
# but since tab isn't always eight spaces, we treat this as unsolveable :(
# there is no common leading whitespace since "      " != "\t"
cat(dedent("

```

```

        another multi-line string, this time with no common leading whitespace
    \tanother multi-line string, this time with no common leading whitespace
    "), sep = "\n")

```

---

|                |                         |
|----------------|-------------------------|
| delayedAssign2 | <i>Delay Evaluation</i> |
|----------------|-------------------------|

---

## Description

delayedAssign2 creates a *promise* to evaluate the given expression if its value is requested.

## Usage

```

delayedAssign2(x, value, eval.env = parent.frame(1),
  assign.env = parent.frame(1), evaluated = FALSE)

```

## Arguments

|            |   |
|------------|---|
| x          | a variable name (given as a quoted string in the function call) |
| value      | an expression to be assigned to x                               |
| eval.env   | an environment in which to evaluate value                       |
| assign.env | an environment in which to assign x                             |
| evaluated  | logical; should value be evaluated before making the promise?   |

## Details

This function is built upon [delayedAssign](#), with the extra argument evaluated. This is helpful in situations where an expression has already been grabbed with substitute somewhere else that you now wish to make into a promise.

While not implemented yet, this is used for the lazy default evaluation in [ArgumentParser](#).

## Value

NULL invisibly.

## Examples

```

# this is a simplified version of what ArgumentParser does with its arguments.
# ArgumentParser hides many details that should be shown here

```

```

# make a list like ArgumentParser
args <- list()
add.argument <- function (name, default)
{
  args[[length(args) + 1]] <-
    list(name = name, default = substitute(default))
  invisible()
}

```

```

# these would normally be added by 'ArgumentParser()' $add.argument'

```

```

add.argument("--alpha", TRUE      )
add.argument("--beta" , `--alpha`)
add.argument("--gamma", `--alpha`)

# this would normally be the environment
# returned by 'ArgumentParser()'$parse.args'
value <- new.env()

for (n in seq_along(args)) {

  # we have 'evaluated = TRUE' here because we don't want 'args[[n]]$default'
  # to be the expression of the promise, but whatever 'args[[n]]$default'
  # evaluates to
  delayedAssign2(args[[n]]$name, args[[n]]$default,
    eval.env = value, assign.env = value,
    evaluated = TRUE)

  # this part would normally be more complex to deal with things like
  # multiple names for 1 argument, 'type', 'choices', and, of course,
  # providing arguments, but i only want to demonstrate 'delayedAssign2' here
}

# then we force evaluate each argument
for (n in seq_along(args)) {
  get(args[[n]]$name, envir = value, inherits = FALSE)
}

# these are all TRUE, could be all NA or FALSE if argument '--alpha=NA' or
# '--alpha=FALSE' was provided to the ArgumentParser
print(as.list(value, all.names = TRUE))

```

---

do.while

*Do While/Until Loops in R*


---

## Description

Allows for the do while/until loop syntax seen in other programming languages.

## Usage

```

expr %while% cond
expr %until% cond

```

## Arguments

|      |  |
|------|--|
| cond | A length-one logical vector that is not NA. Other types are coerced to logical if possible, ignoring any class. <b>MUST</b> be wrapped with parenthesis. |
|------|--|

`expr`                    An *expression* in a formal sense. This is either a simple expression or a so-called *compound expression*, usually of the form { `expr1` ; `expr2` }. **MUST** be wrapped with `do ( . . )`.

### Details

First, the code block `expr` is evaluated, and then the condition `cond` is evaluated. This repeats while/until the condition is TRUE. This contrasts from a [while](#) loop where the condition is evaluated before the code block.

Do while/until loops are implemented using a [repeat](#) loop. That is to say,

```
do(expr) %while% (cond)
do(expr) %until% (cond)
```

are **IDENTICAL** to

```
repeat {
  expr
  if (cond) {
  }
  else break
}
repeat {
  expr
  if (cond)
    break
}
```

, respectively. [break](#), [next](#), and [return](#) statements will behave identically as well.

### Value

NULL invisibly.

### See Also

[repeat](#), [break](#), [next](#)

[Do while loop](#)

### Examples

```
# Suppose you want a unique name for a temporary file (we'll ignore that
# `tempfile` exists for now). We can use a do while loop to create a new
# random name until a unique name is found.
```

```
do ({
  value <- sprintf("%x", sample.int(2147483647L, 1L))
  print(value)
}) %while% (file.exists(value))
```

```
## note that the following is equivalent:
```

```
# do ({
#   value <- sprintf("%x", sample.int(2147483647L, 1L))
#   print(value)
# }) %until% (!file.exists(value))
```

```

# Suppose you want a random number that is greater than one million (we'll ignore
# that `stats::runif` exists for now).
do ({
  value <- sample.int(1.01e+06, 1L)
  print(value)
}) %until% (value > 1e+06)

## note that the following is equivalent:
# do ({
#   value <- sample.int(1.01e+06, 1L)
#   print(value)
# }) %while% (value <= 1e+06)

# Finally suppose you wanted to ask the user for input, but wanted to make sure
# it was valid. Here, we'll say the input is valid if it is all numeric
# characters (ignore leading and trailing whitespace).
# Let's put a limit on it too, only ask a certain amount of times
count <- 0L
do ({
  value <- readline("Enter an integer: ")
  value <- gsub("^\\s+|\\s$", "", value)
  valid <- grepl("^[[:digit:]]+$", value)
  count <- count + 1L
}) %until% (count >= 5L || valid)
print(list(value = value, valid = valid, count = count))

## note that the following is equivalent:
# do ({
#   value <- readline("Enter an integer: ")
#   value <- gsub("^\\s+|\\s$", "", value)
#   valid <- grepl("^[[:digit:]]+$", value)
#   count <- count + 1L
# }) %while% (count < 5L && !valid)

```

---

envvars

---

*Environment Variables*


---

## Description

Get, set, or remove environment variables.

## Usage

```
envvars(...)
```

```
getEnvvar(x, default = NULL)
```

## Arguments

|     |   |
|-----|---|
| ... | environment variables to get, set, or remove. See <b>Examples</b> . |
| x   | a character string holding an environment variable name.            |

`default` if the specified environment variable is not set, this value is returned. This facilitates retrieving an option and checking whether it is set and setting it separately if not. Note that this is converted to a character string.

## Details

`envvars` provides a mechanism for getting, setting, and removing environment variables in a way more in line with [options](#).

Invoking `envvars()` with no arguments returns a list with the current values of the environment variables. To access the value of a single environment variable, one should use, e.g., `getEnvvar("HOME")` rather than `envvars("HOME")` which is a *list* of length one.

## Value

For `getEnvvar`, the current value set for environment variable `x`, or `default` (which defaults to `NA`) if the option is unset.

For `envvars()`, a list of all set environment variables sorted by name. For `envvars(name)`, a list of length one containing the set value, or `NA` if it is unset. For uses setting one or more environment variables, a list with the previous values of the environment variables changed (returned invisibly).

## See Also

[Environment Variables](#)

## Examples

```
# this example runs incredibly weird using 'example', best
# to copy and paste this text within the R console and run
# it from there
## Not run:
oenv <- envvars(); utils::str(oenv) # oenv is a named list

getEnvvar("PATH") == envvars()$PATH # the latter is slower, needs more memory

# change the language, and save the previous value
old.env <- envvars(LANGUAGE = "nn")
old.env          # previous value
envvars("LANGUAGE") # current value

# restore LANGUAGE back to its previous value,
# or remove it if it previously did not exist
envvars(old.env)

envvars(oenv) # reset (all) initial environment variables
envvars("LANGAUGE")

## End(Not run)
```

---

|           |                           |
|-----------|---------------------------|
| file.open | <i>Open a File or URL</i> |
|-----------|---------------------------|

---

**Description**

Opens a file or URL in the user's preferred application.

**Usage**

```
file.open(file)
```

**Arguments**

|      |                           |
|------|---------------------------|
| file | file or URL to be opened. |
|------|---------------------------|

**Details**

file may be any number of files or URLs. Files will be normalized before opening, though they should already be full paths.

On macOS, files and URLs will be open using open, and under any other Unix-alike, files and URLs will be open using xdg-open.

**Value**

file invisibly.

**See Also**

For Windows only, see ?base::shell.exec(file)

**Examples**

```
## Not run:
file.open(tempdir())
file.open("https://cran.r-project.org/doc/manuals/R-exts.html#Documenting-packages")

## End(Not run)
```

---

|           |   |
|-----------|---|
| flat.list | <i>Combine Values into a Flat List (a List with no List Elements)</i> |
|-----------|---|

---

**Description**

Construct a flat list (a list with no list elements).

**Usage**

```
flat.list(...)
```



**Arguments**

... objects, possible named.

**Details**

`rapply` is used to find each non-list element, then is assigned into the next element of a list. As the non-list elements are being recursed through, the names are taken and assigned to the return list.

**Value**

A list.

**Examples**

```
x <- list(a = 1:5, list(list(b = 6:10), c = 11:15), list(d = exp(-4)))
print(x)
flat.list(x)
flat.list(x, e = "testing")
```

---

GeneralizedExtremeValue

*The Generalized Extreme Value Distribution*

---

**Description**

Density, distribution function, quantile function and random generation for the generalized extreme value distribution with location equal to location, scale equal to scale and shape equal to shape.

**Usage**

```
dgev(x, location = 0, scale = 1, shape = 0, log = FALSE)
pgev(q, location = 0, scale = 1, shape = 0, lower.tail = TRUE, log.p = FALSE)
qgev(p, location = 0, scale = 1, shape = 0, lower.tail = TRUE, log.p = FALSE)
rgev(n, location = 0, scale = 1, shape = 0)
```

**Arguments**

|                         |   |
|-------------------------|---|
| <code>x, q</code>       | vector of quantiles.  |
| <code>p</code>          | vector of probabilities.  |
| <code>n</code>          | number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required. |
| <code>location</code>   | vector of locations.  |
| <code>scale</code>      | vector of scales.   |
| <code>shape</code>      | vector of shapes.   |
| <code>log, log.p</code> | logical; if TRUE, probabilities <code>p</code> are given as <code>log(p)</code> .                         |
| <code>lower.tail</code> | logical; if TRUE (default), probabilities are $P[X \leq x]$ otherwise, $P[X > x]$ .                       |

## Details

If location, scale or shape are not specified they assume the default values of 0, 1 and 0, respectively.

The generalized extreme value distribution with location  $\mu$ , scale  $\sigma > 0$  and shape  $\xi$  has density

$$f(s) = (1 + \xi s)^{-1} \left(1 + \frac{1}{\xi}\right) e^{-((1 + \xi s)^{-\frac{1}{\xi}})}$$

for  $1 + \xi s > 0$  where  $s = \frac{(x-\mu)}{\sigma}$ . In the limit  $\xi \rightarrow 0$ , the density simplifies to

$$f(s) = \exp(-s) \exp(-\exp(-s))$$

## Value

dgev gives the density, pgev gives the distribution function, qgev gives the quantile function, and rgev generates random deviates.

The length of the result is determined by n for rgev, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

For scale = 0 this gives the limit as

## Examples

```
shapes <- expression(-1/2, 0, +1/2)
legend.text <- as.expression(lapply(shapes, function(shape) {
  call("=", as.symbol("xi"), shape)
}))
shapes <- vapply(shapes, base::eval, 0)
cols <- c("green3", "red", "blue")
x <- seq.int(-4, 4, length.out = 1001)

# we use plapply here instead of lapply because
# plapply allows us to name the looping arguments
y <- essentials::plapply(
  list(shape = shapes),
  essentials::dgev,
  x = x
)
graphics::par(mar = c(4.9, 4.5, 2.1, 0.4))
graphics::plot(
  xlim = range(x), ylim = range(y),
  panel.first = graphics::grid(col = "gray69"),
  x = NA_real_, y = NA_real_,
  xlab = "x", ylab = ~f(list(x, mu, sigma, xi)),
  main = "Probability density function",
  bty = "L"
)
for (i in seq_along(y)) {
  graphics::lines(x, y[[i]], col = cols[[i]], lwd = 2)
}
graphics::legend(
```

```

      x = "topleft",
      legend = legend.text,
      col = cols,
      lwd = 2,
      bty = "n"
    )
    graphics::title(sub = ~"All with" ~ list(mu == 0, sigma == 1), adj = 1)

y <- essentials::plapply(
  list(shape = shapes),
  essentials::pgev,
  q = x
)
graphics::plot(
  xlim = range(x), ylim = range(y),
  panel.first = graphics::grid(col = "gray69"),
  x = NA_real_, y = NA_real_,
  xlab = "x", ylab = ~F(list(x, mu, sigma, xi)),
  main = "Cumulative probability function",
  bty = "L"
)
for (i in seq_along(y)) {
  graphics::lines(x, y[[i]], col = cols[[i]], lwd = 2)
}
graphics::legend(
  x = "topleft",
  legend = legend.text,
  col = cols,
  lwd = 2,
  bty = "n"
)
graphics::title(sub = ~"All with" ~ list(mu == 0, sigma == 1), adj = 1)

```

hcl.colors2

*Color Palettes***Description**

Create a vector of  $n$  contiguous colors.

**Usage**

```
hcl.colors2(n, palette = "viridis", start = 0, end = if (palette %in%
  special.palettes) (n - 1)/n else 1, alpha, rev = FALSE)
```

```
inferno.colors(n, start = 0, end = 1, alpha, rev = FALSE)
```

```
plasma.colors(n, start = 0, end = 1, alpha, rev = FALSE)
```

```
viridis.colors(n, start = 0, end = 1, alpha, rev = FALSE)
```

```
gg.colors(n, start = 0, end = (n - 1)/n, alpha, rev = FALSE)
```

```
show.colors(x)
```

### Arguments

|                      |   |
|----------------------|---|
| <code>n</code>       | the number of colors ( $\geq 1$ ) to be in the palette.   |
| <code>palette</code> | a valid palette name (one of <code>hcl.pals()</code> ). The name is matched to the list of available palettes, ignoring upper vs. lower case, spaces, dashes, etc. in the matching. |
| <code>alpha</code>   | an alpha-transparency level in the range [0,1] (0 means transparent and 1 means opaque).  |
| <code>rev</code>     | logical indicating whether the ordering of the colors should be reversed.   |
| <code>start</code>   | a number in the range [0,1] at which the vector of colors begins.   |
| <code>end</code>     | a number in the range [0,1] at which the vector of colors ends.   |
| <code>x</code>       | a vector of colors to display.  |

### Details

`hcl.colors2` is a variant of `hcl.colors` that adds a `start` and `end` argument. `hcl.colors2` is slightly faster than `hcl.colors`, though it is not by much.

### Value

For `show.colors`, its argument.

For all others, a character vector of `n` hex color codes.

### Examples

```
show.colors(hcl.colors2(256, palette = "Spectral"))
```

```
show.colors(inferno.colors(256))
show.colors(plasma.colors(256))
show.colors(viridis.colors(256, end = 0.9))
```

```
show.colors(gg.colors(256))
```

---

hypot

*Hypotenuse*


---

### Description

`hypot` returns the “hypotenuse” of all the values present in its arguments.

`phyphot` returns the parallel “hypotenuse” of the input values. It takes any number of vectors as arguments, recycle them to common length, and return a single vector giving the ‘parallel’ “hypotenuse” of the argument vectors.

**Usage**

```
hypot(..., na.rm = FALSE)
phypot(..., na.rm = FALSE)
```

**Arguments**

... numeric or complex arguments.  
 na.rm a logical indicating whether missing values should be removed.

**Details**

The hypotenuse is the longest side of a right-angled triangle, the side opposite the right angle. The length of the hypotenuse is defined as

$$\sqrt{(x^2 + y^2)}$$

The 3-dimensional “hypotenuse” is defined as

$$\sqrt{(x^2 + y^2 + z^2)}$$

The n-dimensional “hypotenuse” is defined as

$$\sqrt{\sum_{k=1}^n x_k^2}$$

Suppose we have a vector  $x$  and we want to know its “hypotenuse”.

If any of  $x$  is infinite, the “hypotenuse” is always Inf.

If `na.rm = FALSE` and any of  $x$  is NA or NaN, the “hypotenuse” is NaN.

Otherwise, the “hypotenuse” will be calculated using the above definition. If `na.rm = TRUE` all NA and NaN values are treated as  $\emptyset$ .

**Value**

For `hypot` a numeric vector of length 1.

For `phypot` a numeric vector. If any of the input values is a zero-length vector the result has length zero. Otherwise, the result has length equal to the length of the longest vector. The rules for determining the attributes of the result are rather complicated. Attributes are only copied from input values whose lengths are equal to the length of the result. If any such input values have a `dim` attribute, the first `dim` attribute is copied to the result. `dimnames` are copied in a similar manner (but only *after* the result has a `dim` attribute). If any such input values have a *conformable* `dimnames` attribute, the first conformable `dimnames` attribute is copied to the result. If a `dim` attribute has *not* been assigned to the result, then finally `names` are copied in a similar manner. If any such input values have a `names` attribute, the first `names` attribute is copied to the result. A result can have a `dim` attribute, a `names` attribute, neither, but cannot have both. `dim` has priority over `names` (similar to *Arithmetic* operators).

**Note**

‘Numeric’ arguments are vectors of type integer and numeric, and logical (coerced to integer). NULL is accepted as equivalent to `numeric(0)`.

**Examples**

```
## when a side is infinite, the hypotenuse is Inf
hypot(Inf, NaN) # Inf
hypot(-Inf, NaN) # Inf (applies to negative infinity too)

## when a side is NA or NaN, the hypotenuse is NaN
hypot(NaN, 0) # NaN
hypot(NA, 0) # NaN

## numbers whose squares would overflow normally are handled well
hypot(.Machine$double.xmax, 5)
hypot(1e+300, 1e+300)

## hypotenuse
hypot(3, 4) # 5
hypot(3+4i) # 5 (works for complex numbers as well)

## 3-dimensional "hypotenuse"
hypot(3, 4, 12) # 13

## n-dimensional "hypotenuse"
hypot(1:100)

x <- seq.int(-3, 3, length.out = 101)
y <- 1
(h <- phypot(x, 1)) # parallel hypotenuse
graphics::plot(
  panel.first = graphics::grid(col = "gray69"),
  x = x, y = h, type = "l",
  main = "Distance from" ~ (list(0, 0)) ~ "to" ~ (list(x, 1))
)
```

---

InverseDistanceWeighting

*Inverse Distance Weighting*


---

**Description**

Interpolate values in n-th dimensional space.

**Usage**

```
IDW(x0, u0, x, p = 2, na.rm = FALSE)
```

**Arguments**

|                    |  |
|--------------------|--|
| <code>x0, x</code> | numeric matrix of coordinates. Each row is a new coordinate, and each column is a new dimension. |
| <code>u0</code>    | numeric or complex vector; known values corresponding to the coordinates of <code>x0</code> .    |
| <code>p</code>     | a positive number; the influence of closer points.   |

na.rm                    logical. Should missing values (including NaN) in `u0` and rows containing missing values in `x0` be removed?

### Details

`x0` and `x` must have the same number of dimensions.

### Value

a numeric or complex vector, equal in length to the number of rows of `x`

### Examples

```
x0 <- c(0, 1, 4, 5)
u0 <- c(1, 2, 2, 1)
x <- seq.int(-4, 9, length.out = 1001)
u <- IDW(x0, u0, x)
graphics::plot(
  panel.first = graphics::grid(col = "gray69"),
  x, u, type = "l", col = "blue", lwd = 2
)
graphics::points(x0, u0, pch = 16, cex = 1.5)
```

---

|                   |                         |
|-------------------|-------------------------|
| legend.dimensions | <i>Size of a Legend</i> |
|-------------------|-------------------------|

---

### Description

This function is used to determine the size of a legend as it appears on a plot.

### Usage

```
legend.dimensions(expr, envir = parent.frame(),
                  enclos = if (is.list(envir) || is.pairlist(envir))
                             parent.frame() else baseenv(),
                  trace = FALSE)
```

### Arguments

|                     |  |
|---------------------|--|
| <code>expr</code>   | a <a href="#">call</a> , an <a href="#">expression</a> of length 1, or a <a href="#">formula</a> of length 2.  |
| <code>envir</code>  | the <a href="#">environment</a> in which <code>expr</code> is to be evaluated. May also be NULL, a list, a data frame, a pairlist or an integer as specified to <a href="#">sys.call</a> .   |
| <code>enclos</code> | Relevant when <code>envir</code> is a (pair)list or a data frame. Specifies the enclosure, i.e., where R looks for objects not found in <code>envir</code> . This can be NULL (interpreted as the base package environment, <a href="#">baseenv()</a> ) or an environment. |
| <code>trace</code>  | Should <code>legend.dimensions</code> print debugging messages?  |

### Details

`expr` would usually be a call to [graphics::legend](#), though anything that adds a legend to a plot and returns identical output to `graphics::legend` is accepted.

If `expr` is a formula, it should be of the form `~call( ... )` where `call( ... )` is the call to be evaluated (the call that produces the legend when evaluated). The [environment](#) of `expr` will be used instead of `envir`, and `enclos` will be ignored.

**Value**

A list with components

w,h positive numbers giving width and height of the legend's box *IN INCHES!*

**Examples**

```
# the expression that produces the desired legend
expr <- ~graphics::legend(

  # the top-left corner of the legend will appear in the
  # top-right corner of the plot
  x = essentials::fix.xlog(graphics::par("usr")[2L]),
  y = essentials::fix.ylog(graphics::par("usr")[4L]),

  legend = letters[1:5], fill = 1:5,
  xpd = TRUE
)

# we'll start by drawing a plot with a legend without
# adjusting the margins. the margins will likely look too
# small or too large
graphics::plot(1:5) ; essentials::add.legend(expr)

# now, we'll adjust the margins using 'legend.dimensions'
# capture the dimensions of the resultant legend
ld <- essentials::legend.dimensions(expr)

essentials::adj.margins(ld)
graphics::plot(1:5) ; essentials::add.legend(expr)

# now, we'll adjust the legend such that it is centered
# vertically
essentials::location(expr) <- essentials::location(ld, adj = 0.5)
graphics::plot(1:5) ; essentials::add.legend(expr)

# you do not need to use 'legend.dimensions' explicitly, you
# could supply 'expr' directly to 'adj.margins' and
# 'location'
essentials::adj.margins(expr)
```



```
essentials::location(expr) <- essentials::location(expr, adj = 0.75)
graphics::plot(1:5) ; essentials::add.legend(expr)
```

---

|             |   |
|-------------|---|
| list.files2 | <i>List the Files in a Directory/Folder</i> |
|-------------|---|

---

## Description

Unlike [list.files](#), paths will not be converted to the native encoding, but instead to UTF-8.

In Windows, `list.files2` will use [python](#), so make sure you have it installed.

Under Unix-alikes, `list.files2` will just call [list.files](#).

## Usage

```
list.files2(path = ".", pattern = NULL, all.files = FALSE,
            full.names = FALSE, recursive = FALSE,
            ignore.case = FALSE, include.dirs = FALSE, no.. = FALSE)

dir2(path = ".", pattern = NULL, all.files = FALSE,
     full.names = FALSE, recursive = FALSE,
     ignore.case = FALSE, include.dirs = FALSE, no.. = FALSE)
```

## Arguments

`path`, `pattern`, `all.files`, `full.names`, `recursive`, `ignore.case`, `include.dirs`, `no..`.  
See [list.files](#).

## Value

A character vector containing the names of the files in the specified directories.

## See Also

[list.files](#)

## Examples

```
## Not run:
FILE <- paste0(tempfile(tmpdir = tempfile("dir")), "_testing_\u{03B4}.txt")
dir.create(dirname(FILE))
invisible(file.create(FILE))

x <- list.files (dirname(FILE), full.names = TRUE)
y <- list.files2(dirname(FILE), full.names = TRUE)
`names<-`(file.exists(x), x)
`names<-`(file.exists(y), y)

unlink(dirname(FILE), recursive = TRUE)

## End(Not run)
```

listify

*Create a List from an Object***Description**

Coerce an object to a list.

**Usage**

```
listify(x)
```

**Arguments**

`x` an object.

**Details**

`listify` is mostly used in combination with `mapply` or `.mapply`. For the dots argument of `.mapply`, each element of which is looped over, `listify` is a convenient way to turn each element into an iterable version of itself.

**Value**

A list.

**Note**

`listify` has unusual behaviour for `data.frame` objects.

An S4 `data.frame` (created by `x <- new("data.frame")`) will return `x`. This is because `inherits(x, "list")` for an S4 object `x` uses `extends(class(x), "list")` to determine if `x` inherits from "list" (which it does for and S4 `data.frame`).

An unmodified S3 `data.frame` (created by `x <- data.frame()`) will return `list(x)`. This is because `inherits(x, "list")` for an S3 object `x` with a `class` attribute uses `"list" %in% oldClass(x)` to determine if `x` inherits from "list" (which it does not in this case).

An S3 `data.frame` could be modified (such as `x <- structure(data.frame(), class = c("data.frame", "list"))`) such that `listify` would return `x`. Where `"list" %in% oldClass(x)` was FALSE before, it is TRUE here.

**Examples**

```
listify(5)          # when 'x' is not a list, returns 'list(x)'
listify(list(5))    # when 'x' is a list, returns 'x'
```

```
## when 'x' is an S4 data.frame, returns 'x'
listify(methods::new("data.frame"))
```

```
## when 'x' is an unmodified S3 data.frame, returns 'list(x)'
listify(data.frame())
```

```
## S3 data.frame 'x' could be modified such that 'listify' returns 'x'
listify(structure(data.frame(), class = c("data.frame", "list")))
```

Missing

*Does an Argument have a Value?*

## Description

Missing can be used to test whether a value is the missing argument.

## Usage

```
Missing(x, name)
```

## Arguments

|      |  |
|------|--|
| x    | an object of class "ParsedArgs".   |
| name | character string or symbol, the name of the argument to test for missing-ness. |

## Details

Missing will return FALSE when an argument has a value, even if it wasn't provided. See **Examples**.

Missing does not evaluate its argument name, similar to missing.

## Value

TRUE or FALSE

## Examples

```
parser <- essentials::ArgumentParser()
parser$add.argument("--arg1")
parser$add.argument("--arg2", default = "def")
pargs <- parser$parse.args()

list(
  arg1 = essentials::Missing(pargs, arg1),
  arg2 = essentials::Missing(pargs, "arg2")
)

# with R >= 4.1.0, use the forward pipe operator `|>` to
# make calls to `Missing` more intuitive:
# list(
#   arg1 = pargs |> essentials::Missing(arg1),
#   arg2 = pargs |> essentials::Missing("arg2")
# )
```

---

|                  |   |
|------------------|---|
| normalizeAgainst | <i>Normalize File Paths Against a Different Directory</i> |
|------------------|---|

---

### Description

Convert file paths to canonical form, using a different working directory. By default, the different working directory would be the executing script's directory.

### Usage

```
normalizeAgainst(..., against = this.dir(verbose = FALSE))
```

### Arguments

|         |   |
|---------|---|
| ...     | arguments passed to <a href="#">normalizePath()</a> .   |
| against | character string (or possibly NULL). The directory against which file paths will be normalized. |

### Details

When against is a character string, the working directory is changed to against, then the paths are [normalized](#), and then the working directory is changed back to its previous value.

### Value

A character vector, see [normalizePath](#).

---

|         |                       |
|---------|-----------------------|
| numbers | <i>Number Vectors</i> |
|---------|-----------------------|

---

### Description

Creates or coerces objects of type “numeric” or “complex”. `is.numbers` is a more general test of an object being interpretable as numbers.

### Usage

```
numbers(length = 0)
as.numbers(x, ...)
is.numbers(x)

## Default S3 method:
as.numbers(x, strict = TRUE, ...)
```

## Arguments

|        |  |
|--------|--|
| length | A non-negative integer specifying the desired length. Double values will be coerced to integer: supplying an argument of length other than one is an error.  |
| x      | object to be coerced or tested.  |
| strict | TRUE or FALSE, should a vector of complex numbers where each element is strictly real (real component is NA or NaN or imaginary component is NA or NaN or 0) be converted to a vector of real numbers? |
| ...    | further arguments passed to or from other methods.   |

## Details

numbers is identical to [numeric](#) and [double](#) (and real). It creates a double-precision vector of the specified length with each element equal to 0.

as.numbers attempts to coerce its argument to be of double or complex type: like [as.vector](#) it strips attributes including names.

is.numbers is a more general test of an object being considered numbers, meaning the base type of the class is double or integer or complex *and* values can reasonably be regarded as numbers (e.g., arithmetic on them makes sense, and comparison should be done via the base type).

## Value

for numbers see [double](#).

as.numbers returns either a double or complex vector.

is.numbers returns TRUE if its argument is of [mode](#) "numeric" or "complex" and not a factor, and FALSE otherwise.

## Examples

```
x <- 1:5
names(x) <- c("a", "b", "c", "d", "e")
as.numbers(x) # vector converted from integer to double, names removed

x <- x + 0i # x is now a complex vector
as.numbers(x) # vector of type double since all numbers were purely real

## vector of type complex, despite being purely real
as.numbers(x, strict = FALSE)

x <- x + 1i
## vector remains of type complex since numbers are not purely real
as.numbers(x)
```

---

|               |                        |
|---------------|------------------------|
| numbers-class | <i>Class "numbers"</i> |
|---------------|------------------------|

---

### Description

An umbrella formal class encompassing all objects interpretable as numbers. This includes integer, double and complex.

### Extends

Class "[numeric](#)", directly. Class "[complex](#)", directly.

### Methods

**coerce** A method is defined to coerce an arbitrary object to a numbers vector by calling [as.numbers](#). The object is returned as is if it already extends class "numeric" or "complex".

---

|            |   |
|------------|---|
| parse.args | <i>Parse the Command-Line Arguments into an Environment</i> |
|------------|---|

---

### Description

Parse any series of arguments into an environment using an argument parser.

### Usage

```
## S4 method for signature 'essentials_ArgumentParser'
parse.args(args = Args(),
  warnPartialMatchArgs = getOption("warnPartialMatchArgs", FALSE))
```

### Arguments

**args** character vector. The arguments to parse into an environment.

**warnPartialMatchArgs** logical. If true, warns if partial matching is used in argument matching.

### Details

When args is not provided, the arguments are provided from [commandArgs\(\)](#), but only if the executing script was run from the command-line // terminal.

### Value

an object of class "ParsedCommandArgs".

---

|               |                            |
|---------------|----------------------------|
| path.contract | <i>Contract File Paths</i> |
|---------------|----------------------------|

---

## Description

Contract a path name, for example by replacing the user's home directory (if defined on that platform) with a leading tilde.

## Usage

```
path.contract(path, ignore.case = .Platform$OS.type == "windows")
```

## Arguments

|             |   |
|-------------|---|
| path        | character vector containing one or more path names.                             |
| ignore.case | TRUE or FALSE, should case be ignored when replacing the user's home directory? |

## Details

On Windows, paths are case insensitive, so something like 'C:\Users\effective\_user\Documents' is regarded as equivalent to 'c:\users\effective\_user\documents'.

Under Unix-alikes, paths are case sensitive, so something like '/home/effective\_user' is not equivalent to '/HOME/effective\_user'.

By default, path.contract respects each of these behaviours, but can be changed if desired.

Additionally, on Windows, the file separator may be either "/" or "\", for example 'C:\Users\effective\_user\Documents' and 'C:/Users/effective\_user/Documents' would be treated as equivalent (or any mixture of the two, for example 'C:/Users\effective\_user/Documents').

The 'path names' need not exist nor be valid path names.

## Value

A character vector of possibly contracted path names: where the home directory is unknown of none is specified the path is returned unchanged.

## Note

Using ./, ../, symbolic links, hard links, or multiple mounts will *NOT* be resolved before attempting to contract path names. If you wish to resolve these before contracting path names, see [normalizePath](#).

## See Also

[path.expand](#), [basename](#), [normalizePath](#), [file.path](#).

## Examples

```
stopifnot(path.contract(path.expand(x <- c("~/", "~/foo")))) == x)
# Note that this is not necessarily true the other way around (in Windows)
# simply because the path separator may have changed

tilde <- path.expand("~/")
if (tilde == "~/") {
  cat("the home directory is unknown or none is specified\n")
} else {
  paths <- file.path(c(tilde, toupper(tilde), tolower(tilde)), "foo")
  print(cbind(
    Path = paths,
    `Contracted Path` = path.contract(paths),
    `Contracted Path (ignoring case)` = path.contract(paths, ignore.case = TRUE),
    `Contracted Path (with case)` = path.contract(paths, ignore.case = FALSE)
  ), quote = FALSE)
}
```

---

plapply

*Apply a Function to Multiple List or Vector Arguments*


---

## Description

plapply, psapply, and pvapply are multivariate (parallel) versions of [lapply](#), [sapply](#), and [vapply](#). They take any number of vectors as arguments, recycle them to common length, and return a single vector.

## Usage

```
plapply(X, FUN, ...)
```

```
psapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

```
pvapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

## Arguments

|           |  |
|-----------|--|
| X         | list of arguments to vectorize over.   |
| FUN       | function to apply, found via <a href="#">match.fun</a> .   |
| ...       | optional arguments to FUN.   |
| simplify  | logical or character string; should the result be simplified to a vector, matrix, or higher dimensional array if possible? The default value, TRUE, returns a vector or matrix if appropriate, whereas if simplify = "array" the result may be an <a href="#">array</a> of higher dimension. |
| USE.NAMES | logical; if TRUE and one of X is character and of equal length to the result, use that element of X as <a href="#">names</a> for the result unless it had names already.   |
| FUN.VALUE | a (generalized) vector; a template for the return value from FUN.  |



## Details

In `plapply`, after the arguments have been recycled to common length, a list is made with length equal to the common length. The value returned in the  $j$ -th position of the list is the function `FUN` applied to the  $j$ -th elements of `X`, that is:

```
value[[j]] <- FUN(X[[1L]][[j]], X[[2L]][[j]], ...)
```

with more or less `X[[i]][[j]]` depending on the length of `X`. Additionally, the arguments in the call will be named if `X` has names.

Unlike `lapply`, `X` and its elements will not be coerced by `base::as.list`. Instead, their subsetting (`[[`), `length`, `lengths`, and `names` methods will be used.

Simplification in `psapply` and `pvapply` is the same as `sapply` and `vapply`.

## Value

For `plapply` a list. If any of the input values is a zero-length vector the result has length zero. Otherwise, the result has length equal to the length of the longest vector. Names are copied from input values where possible. Names are only copied from input values whose lengths are equal to the length of the result. If any such input values have a `names` attribute, the first `names` attribute is copied to the result.

For `psapply` a list, the same as `plapply`. If `USE.NAMES` and `plapply` did not give names to its return value, then names are added where possible. If the input values contain a character vector whose length is equal to the length of the result, the names of the the result will be said character vector. Then the result may be simplified to a matrix // array via `simplify2array`.

For `vapply` a vector or array of type matching the `FUN.VALUE`. If `length(FUN.VALUE) == 1` a vector of the common length of `X` is returned, otherwise an array. If `FUN.VALUE` is not an `array`, the result is a matrix with `length(FUN.VALUE)` rows and common length of `X` columns, otherwise an array with `dim c(dim(FUN.VALUE), <common length of X>)`

The `(Dim)names` of the array value are taken from the `FUN.VALUE` if it is named, otherwise from the result of the first function call. Column names of the matrix or more generally the names of the last dimension of the array value or names of the vector value are set from `X` as in `psapply`.

## Note

`plapply` is based on `mapply`. The important differences are:

**Argument FUN** `plapply` has `FUN` as the second argument (same as `lapply`) while `mapply` has `FUN` as the first argument.

**Arguments to Vectorize Over** `plapply` has `X` as the first argument which is a collection of arguments to vectorize over (same as `lapply`, aside from the multivariate aspect) while `mapply` has `...` as the collection of arguments to vectorize over. This means you'll need to use `do.call` or similar if your collection of arguments to vectorize over are already stored in a list, pretty inconvenient.

**Optional // Other // Further Arguments to FUN** `plapply` has `...` immediately after argument `FUN` as the additional arguments to provide to `FUN` each time it is called (same as `lapply`) while `mapply` has `MoreArgs` as the list of additional arguments to provide to `FUN`. If your additional arguments are already stored in a list, this makes `plapply` inconvenient to use. However, instead of abandoning `plapply` in favour of `mapply`, consider using `.plapply`, an alternate version that accepts an argument `dots` instead of `...`

**More on Additional Arguments to FUN** When using `mapply`, if `MoreArgs` contains a symbol or call, it will not be properly protected from early evaluation (despite begging R-Core to change it and even offering a patch :/). I consider this problem big enough that I would never use

mapply. It is rare that you'd run into such a scenario, but when you do, it's incredibly hard to debug, and even more annoying to fix while still using mapply. plapply and .plapply have no such issue when ... or dots contains a symbol or call, they will never be evaluated unnecessarily.

### See Also

[.plapply](#)

### Examples

```
plapply(list(
  col = c("red", "green", "blue"),
  cex = c(1, 1.5, 2),
  main = c("title 1", "title 2", "title 3")
), graphics::plot, x = 1:5, pch = 16)
```

---

|             |                      |
|-------------|----------------------|
| progressBar | <i>Progress Bars</i> |
|-------------|----------------------|

---

### Description

Generalized progress bars, with 3 default flavours.

### Usage

```
progressBar(name = names(progressBars), ...)
```

```
getProgress(pb)
setProgress(pb, ...)
```

```
# declare a new progress bar
setProgressBarMethod(name, definition)
```

```
increment(x, ...)
decrement(x, ...)
```

### Arguments

|            |  |
|------------|--|
| name       | character string, which kind of progress bar to make or set a method. By default, one of "tk", "txt", and "win" (Windows exclusive). |
| definition | a function that will create a progress bar corresponding to name.  |
| pb, x      | a progress bar object.   |
| ...        | Further arguments passed to other methods.   |

### Value

For progressBar, a progress bar object.

For getProgress, setProgress, increment, and decrement, a length-one numeric vector giving the previous value (visibly for getProgress).

**See Also**

[tkProgressBar](#), [txtProgressBar](#), and [winProgressBar](#) on Windows.

**Examples**

```
# if we use 'tryCatch', we can make use 'finally' to guarantee the progress bar
# is closed, regardless of signalling an error or user interrupt. not entirely
# useful in this example, but it can be in longer, more complicated situations
tryCatch({
  pb <- progressBar(if (.Platform$OS.type == "windows")
    "win"
  else "txt", max = 20, style = 3)
  for (i in 1:20) {
    Sys.sleep(0.05)
    increment(pb)
  }
}, finally = close(pb))
```

---

pseudoglobalenv

*Pseudo Global Environment*

---

**Description**

Create an environment similar to [globalenv\(\)](#) (with the same parent environments), intended for use with [source\(\)](#).

**Usage**

```
pseudoglobalenv(...)
```

**Arguments**

... further arguments passed to [new.env\(\)](#).

**Value**

a new (empty) environment with the same enclosure as the global environment.

---

python

*Start a New Python Session*

---

**Description**

Start a new Python session with the specified options.

Make sure you have [python](#) installed.

**Usage**

```
python(options = NULL, command = NULL, module = NULL, file = NULL,
  args = NULL, chdir = FALSE, ...)
```

## Arguments

|         |  |
|---------|--|
| options | character vector. The options to be provided for the new Python session.   |
| command | character vector. The expressions to be run in the new Python session, provided to option ‘-c cmd’.                                      |
| module  | character string. The library module to run as a script, provided to option ‘-m mod’.  |
| file    | character string. The filename of the Python script to run in the new Python session, provided to option ‘file’                          |
| args    | Converted to character in a similar manner to <a href="#">withArgs</a> .   |
| chdir   | logical; if TRUE and file is a pathname, the R working directory is temporarily changed to the directory containing file for evaluating. |
| ...     | further arguments passed to <a href="#">system()</a> .   |

## Value

The value of [system\(\)](#). If intern = FALSE, it is returned invisibly.

## Examples

```
## Not run:
essentials::python(command = essentials::dedent(r"{
  print("Hello World!")
  print("An example python script...")
}"))

## End(Not run)
```

---

R

*Start a New R Session*


---

## Description

Start a new R session with the specified options.

## Usage

```
R(options = NULL, file = NULL, exprs = NULL, args = NULL,
  chdir = FALSE, ...)
```

```
Rcmd(options = NULL, command = "", args = NULL, ...)
```

```
Rscript(options = NULL, file = NULL, exprs = NULL, args = NULL,
  chdir = FALSE, ...)
```

```
Rterm(options = NULL, file = NULL, exprs = NULL, args = NULL,
  chdir = FALSE, ...)
```

## Arguments

|         |  |
|---------|--|
| options | character vector. The options to be provided for the new R session. For Rcmd on Windows, options will not do anything.   |
| file    | character string. The filename of the R script to run in the new R session, provided to option '--file='.  |
| exprs   | character vector. The expressions to be run in the new R session, provided to option '-e'.   |
| args    | any R object. The command-line arguments to be provided <i>after</i> '--args', or for Rcmd, arguments provided after command. Converted to character in a similar manner to <a href="#">withArgs</a> . |
| chdir   | logical; if TRUE and file is a pathname, the R working directory is temporarily changed to the directory containing file for evaluating.   |
| ...     | further arguments passed to <a href="#">system()</a> .   |
| command | character string. Naming one of the commands to execute from R CMD usage message.  |

## Details

'Rterm' is an executable only available on Windows, but it is preferable because it has more intuitive quoting rules than 'R' (see [commandQuote](#)). If Rterm is called on a Unix-alike OS, R is called instead.

## Value

The value of [system\(\)](#). If intern = FALSE, it is returned invisibly.

## Examples

```
# if you're on Windows, you should notice that the quoting rules for 'Rterm' are
# far more intuitive than for 'R'
#
# if you're under a Unix-alike, you should notice that 'Rterm' and 'R' do the
# same thing
```

```
essentials::R      (exprs = r"{cat(commandArgs(), sep = "\n")}")
essentials::Rterm  (exprs = r"{cat(commandArgs(), sep = "\n")}")
essentials::Rscript(exprs = r"{cat(commandArgs(), sep = "\n")}")
essentials::Rscript(exprs = r"{cat(commandArgs(), sep = "\n")}", quiet = TRUE)
```

```
essentials::Rcmd(command = "INSTALL", args = "--help")
```

## Description

When providing arguments to an R script, it is often necessary to quote the arguments such that they are interpreted correctly by the shell before being passed to the script. However, this comes with the issue of different quoting conventions for different shells, meaning that the strings will not be interpreted the same between different shells. Additionally, some characters may not be allowed for certain shells.

You can get around this with `writeArgs` which will right your arguments (and comments!) to a file, and returns a filename that can be used in the arguments place.

`readArgs` is the corresponding function to read those arguments back into R.

## Usage

```
writeArgs(x, file = tempfile(pattern = pattern, fileext = fileext),
  pattern = "args", fileext = ".Rargs", comments = TRUE,
  nlines.between.comment.and.args = 0, nlines.between.args = 2,
  at = TRUE, name = NULL)
```

```
readArgs(file, name = NULL)
```

## Arguments

|  |   |
|--|---|
| <code>x</code>                               | any R object. If a list, each element will be turned into a character vector (in the same way as <code>withArgs</code> converts its arguments), otherwise will be turned into a character vector (same as <code>withArgs</code> ). The arguments to be written. |
| <code>file</code>                            | character string or NULL. Name of the file to be read from, or "" to write to <code>stdout()</code> (intended for verifying the output looks as expected), or NULL to return the formatted text without writing anywhere.                                       |
| <code>pattern, fileext</code>                | character string. Allows for easier means of editing the filename created.  |
| <code>comments</code>                        | logical. Should the <code>comments</code> of <code>x</code> be written as well?   |
| <code>nlines.between.comment.and.args</code> | a non-negative integer specifying the number of empty lines between each set of comments and arguments.   |
| <code>nlines.between.args</code>             | a non-negative integer specifying the number of empty lines between each set of arguments. Only used for a list <code>x</code> .  |
| <code>at</code>                              | logical. Should an "@" character be placed before the returned filename?  |
| <code>name</code>                            | The name of the format for reading/writing, see <b>Details</b> . For writing, will be ignored if <code>file</code> is a non-empty character string.   |

## Details

`readArgs` and `writeArgs` accept many formats for reading and writing arguments, based on the file extension.

**R arguments, extension ".Rargs"** Arguments are read as though they were R strings, separated by newline "\n" or semicolon "; ". This means arguments must be quoted and can use any of the escape sequences seen in [Quotes](#). The most reliable method, also the slowest (but not by much). Can contain R comments, will be ignored when read.

**Python arguments, extension ".pyargs"** Arguments are read one per line. No quoting rules apply. Arguments cannot contain newline or carriage return, and will not be read/written properly if so. The fastest method. Cannot contain comments.

**Comma-separated arguments, extension ".csv"** Arguments are read using [scan](#), delimited/separated by comma ",", ". Arguments may be quoted with double quotes "\"", and must be quoted if they contain comma, newline, or double quotes. Arguments cannot contain carriage return. Cannot contain comments.

**Tab-separated arguments, extension ".tsv"** Arguments are read using [scan](#), delimited/separated by tab "\t". Arguments may be quoted with double quotes "\"", and must be quoted if they contain tab, newline, or double quotes. Arguments cannot contain carriage return. Cannot contain comments.

**Other arguments, other extension** Arguments are read using [scan](#), delimited/separated by 'white-space'. Arguments may be quoted with single quotes "'" and double quotes "\"", and must be quoted if they contain white-space, single quotes, or double quotes. Arguments cannot contain carriage return. Quoted arguments cannot contain an odd number of trailing backslashes, or an odd number of backslashes before an embedded quoting character. Can contain comments, marked by a non-quoted hash character "#".

The methods listed above can read/write from compressed files (see [gzfile](#)). For "csv", "tsv", and "other" methods, writeArgs tries to space the text such that the file will look nice to open within a text editor and 'Excel'. Each line of *arguments* (not comments) will have at most 80 characters (to look nice in a text editor) or at most 10 columns (to look nice in 'Excel'), whichever comes first (though this rule is broken by strings with more than 80 characters).

## Value

for file = NULL, the formatted text.

for file = "", the formatted text invisibly.

otherwise, a character string naming a file containing your arguments.

## Examples

```
x <- letters; essentials::writeArgs(x, file = "")

comment(x) <- essentials::dedent("
  adding a comment
  to our arguments
"); essentials::writeArgs(x, file = "")

x <- list(

  local({
    x <- c("\xC5", "\xC9", "\xD8", "\xEC", "\xFC")
    Encoding(x) <- "latin1"
    comment(x) <- "accented characters"
    x
  }),

  local({
```

```

    x <- c("\u{03C3}", "\u{03B4}")
    comment(x) <- 'greek letters (default encoding "UTF-8")'
    x
  }},

  local({
    x <- "fa\xE7ile"
    Encoding(x) <- "latin1" # x is intended to be in latin1
    comment(x) <- essentials::dedent(r"{
      another non-ASCII character ("unknown" in UTF-8 locale,
      "latin1" in IS08859-1, ...)
    }")
    x
  }},

  local({
    x <- c("\u{7B90}", "\u{5316}", "\u{5B57}")
    comment(x) <- 'chinese characters (default encoding "UTF-8")'
    x
  }},

  local({
    x <- c("\U{0001D11E}", "\U{0001D4D7}")
    comment(x) <- "rarer characters outside the usual 16^4 range"
    x
  }},

  local({
    x <- essentials::dedent(r"{
      this would be 'rather' annoying to quote for a `shell` on $Unix$,
      and even "more" so on Windows because of the \"double\" quotes!
    }")
    comment(x) <- "all ASCII characters, annoying to quote, hard to read"
    x
  })
)
comment(x) <- essentials::dedent("
  these are some unusual characters
  but should still behave correctly
")

# the same arguments in different formats
essentials::writeArgs(x, "", name = "Rargs")
essentials::writeArgs(x, "", name = "pyargs")
essentials::writeArgs(x, "", name = "csv")
essentials::writeArgs(x, "", name = "tsv")
essentials::writeArgs(x, "", name = NULL)

FILE <- essentials::writeArgs(x, at = FALSE)
y <- essentials::readArgs(FILE)

```



```

# for the purpose of comparison, we need 'x' to be a character vector
z <- essentials::asArgs(x)

# let's check that the arguments in 'x' match the
# arguments written to and read back from 'FILE' (in 'y')
#
# we don't use 'identical(x, y)' because 'x' has
# attributes (a comment) while 'y' does not
if (length(z) != length(y)) {
  cat(gettextf("Catastrophic failure, wrote %d arguments, read %d\n",
    length(z), length(y)),
    file = stderr())
  stop("Please submit a bug report using ",
    "utils::bug.report(package = \"essentials\")")
} else if (any(i <- z != y)) {
  cat(ngettext(sum(i),
    "The following argument was written or read incorrectly!\n",
    "The following arguments were written or read incorrectly!\n"),
    file = stderr())
  print(z[i])
  cat("\nIncorrectly written or read as:\n", file = stderr())
  print(y[i])
  stop("Please submit a bug report using ",
    "utils::bug.report(package = \"essentials\")")
} else cat("Yay! The arguments were written and read correctly!\n")

# Using writeArgs and withArgs, Rscript
cat(
  essentials::dedent(r"{
    withAutoprint({
      parser <- essentials::ArgumentParser()
      parser$add.argument("args", nargs = "*")
      pargs <- parser$parse.args()
      print(pargs$args)
    })
  }"),
  file = script <- tempfile(), sep = "\n"
)
essentials::withArgs(
  source(script, local = TRUE, echo = FALSE),
  paste0("@", FILE)
)
essentials::Rscript("--default-packages=NULL", script, args = paste0("@", FILE))

# reading/writing from/to a compressed file
FILE2 <- essentials::writeArgs(x, at = FALSE, fileext = ".Rargs.gz")
stopifnot(identical(

```

```

    essentials::readArgs(FILE),
    essentials::readArgs(FILE2)
  ))

# miniscule difference, more desirable with more arguments
c(file.size.original = file.size(FILE),
  file.size.compressed = file.size(FILE2))

```

rowmatch

*Row Matching***Description**

rowmatch and row.match return a vector of the positions of (first) row matches of its first argument in its second.

%rowin% and %row.in% are more intuitive interfaces as binary operator, which return a logical vector indicating if there is a row match or not for its left operand.

**Usage**

```
rowmatch(x, table, nomatch = NA_integer_, incomparables = NULL)
```

```
x %rowin% table
```

```
row.match(x, table, nomatch = NA_integer_, incomparables = NULL)
```

```
## S4 method for signature 'data.frame,data.frame'
```

```
row.match(x, table, nomatch = NA_integer_, incomparables = NULL)
```

```
x %row.in% table
```

```
## specifically for row matching data frames
```

```
row.match.data.frame(x, table, nomatch = NA_integer_, incomparables = NULL)
```

**Arguments**

|               |  |
|---------------|--|
| x             | vector, matrix, array, data.frame or NULL: the row values to be matched.   |
| table         | vector, matrix, array, data.frame or NULL: the row values to be matched against.   |
| nomatch       | the value to be returned in the case when no row match is found. Note that it is coerced to integer.   |
| incomparables | a vector, matrix, array or data.frame of rows that cannot be matched. Any row in x matching a value in this object is assigned the nomatch value. In <a href="#">match</a> , FALSE is equivalent to NULL, the same is true here. |

**Details**

%rowin% and %row.in% are currently defined as "`%rowin%`" <- function(x, table) rowmatch(x, table, nomatch = 0) > 0 "`%row.in%`" <- function(x, table) row.match(x, table, nomatch = 0) > 0

rowmatch is intended for matrix-like objects including vectors, matrices and arrays. Data frames are coerced to arrays.

Similar to `row.names` being generic with `rownames` as the default method, `row.match` is generic with `rowmatch` as the default method.

`row.match.data.frame` is the method used for `row.match` when one of `x`, `table` and `incomparables` is a data frame. It is available incase you want to call it directly. Instead of coercing to arrays, it coerces to data frames.

That `%rowin%` and `%row.in%` never return NA makes them particularly useful in `if` conditions.

## Value

A vector of the same length as the number of rows in `x`.

`rowmatch` and `row.match`: An integer vector giving the row position in `table` of the first row match if there is a row match, otherwise `nomatch`.

If the *i*-th row of `x` is found to equal the *j*-th row of `table` then the value returned in the *i*-th position of the return value is *j*, for the smallest possible *j*. If no match is found, the value is `nomatch`.

`%rowin%` and `%row.in%`: A logical vector, indicating if a row match was located for each row of `x`: thus the values are TRUE or FALSE and never NA.

---

Runge-Kutta Methods      *Runge-Kutta Methods*

---

## Description

Performs the Runge-Kutta method on a collection of dependent variables

## Usage

```
EulerMethod(independent, initialConditions, fun)
RK1(independent, initialConditions, fun)
```

```
ImprovedEulerMethod(independent, initialConditions, fun)
RK2(independent, initialConditions, fun)
```

```
RungeKuttaMethod(independent, initialConditions, fun)
RK4(independent, initialConditions, fun)
```

## Arguments

|                                |   |
|--------------------------------|---|
| <code>independent</code>       | list or numeric; if a list, the first element will be used<br>A numeric sequence of independent variable values<br>The advantage of a list is explained in the <b>Value</b> section |
| <code>initialConditions</code> | numeric or complex; initial values for a series of dependent variables  |
| <code>fun</code>               | function; accepts at least two arguments, the first an independent variable value, the second a series of dependent variable values   |

## Details

Consider a problem given as:

$$\frac{dx}{dq} = f(q, x), x(q[0]) = x[0]$$

Here  $x$  is an unknown function (scalar or vector) of independent quantity  $q$  that we wish to approximate. We know that  $dx/dq$ , the rate at which  $x$  changes, is a function of  $q$  and  $x$  itself. At the initial quantity  $q_0$ , the corresponding value of  $x$  is  $x_0$ . To use any of the Runge-Kutta methods, provide a series of independent quantity values as "independent" (the first of which should be  $q_0$ ), the initial quantities of  $x$ ,  $x_0$ , as "initialConditions", and the function  $f(q, x)$  as "fun"

## Value

The advantage of a list is that a name may be provided to the independent variable. For instance, you could say

```
EulerMethod(independent = seq(0, 1, 0.001), ...)
```

or you could say

```
EulerMethod(independent = list(t = seq(0, 1, 0.001)), ...)
```

This way the return value will name the independent variable

## Examples

```
# Consider a particle moving in a bowl
# We'll use cylindrical coordinates r, theta, and z
#   r      = radial position (distance from z-axis)
#   theta  = angular position
#   z      = height
# The height of the bowl is z = r
# The independent variable (in this case) is time (denoted t)
# After some calculations, the differential equations obtained are
#   d/dt(r^2 * d/dt(theta)) = 0
#   d^2/dt^2(r) = r/2 * (d/dt(theta))^2 - g/2
# where g is the acceleration due to gravity
# This gives us three equations as follows
#   d/dt(r)      = f[r]      = rdot
#   d/dt(rdot)   = f[rdot]   = (r * (d/dt(theta))^2 - g)/2
#   d/dt(theta) = f[theta] = r0^2 * thetadot0 / r^2
# where
#   rdot        = d/dt(r)
#   r0          = initial value of radial position
#   rdot0       = initial value of radial velocity
#   theta0      = initial value of angular position
#   thetadot0   = initial value of angular velocity
# With all of this information, we can define
#   a sequence of time values
#   our initial conditions
#   derivatives of the dependents with respect to the independent

g <- 1
r0 <- 1      # 1 away from the center
rdot0 <- 0   # not moving radially
theta0 <- 0  # in the positive x-direction
thetadot0 <- 0.5 # spinning counter-clockwise with angular speed 0.5
```

```

k <- r0^2 * thetadot0
thetadot <- function(r) k/r^2

independent <- list(t = seq.int(0, 21.6, 0.001))
initialConditions <- c(r = r0, rdot = rdot0, theta = theta0)
fun <- function(independent, dependents) {
  r <- dependents[1L]
  thetadot <- k/r^2
  c(dependents[2L], (r * thetadot^2 - g)/2, thetadot)
}

# finally, simply call
value <- essentials::EulerMethod(independent, initialConditions, fun)

x <- value$r * cos(value$theta)
y <- value$r * sin(value$theta)
xlim <- c(-1, 1) * max(abs(c(x, y)))

graphics::par(mar = c(5.1, 4.1, 0.4, 0.4))
graphics::plot(
  xlim = xlim, ylim = ylim, asp = 1,
  panel.first = grid(col = "gray69"),
  x = x, y = y,
  col = grDevices::hcl.colors(length(x)),
  pch = 16, cex = 0.5,
  bty = "n",
  xlab = "x", ylab = "y"
)
graphics::title(
  sub = bquote(list(
    r[0] == .(r0),
    dot(r)[0] == .(rdot0),
    theta[0] == .(theta0),
    dot(theta)[0] == .(thetadot0)
  )),
  adj = 1
)

# Consider an animal population defined by
#  $d/dt(P) = k * P * (M - P) - h$ 
# where
#  $t$  = time
#  $P$  = population as a function of time

```

```

# k = how fast the population increases
# M = carrying capacity of the population within their environment
# h = amount harvested / / hunted
#
#
# The solution to this equation is
#  $P(t) = \frac{M_1 * (P_0 - M_2) - M_2 * (P_0 - M_1) * \exp(-k * \Delta * t)}{((P_0 - M_2) - (P_0 - M_1) * \exp(-k * \Delta * t))}$  /
# where
#  $P_0$  = initial population
#  $\Delta = \sqrt{M^2 - 4 * h/k}$ 
#  $M_1 = (M + \Delta)/2$ 
#  $M_2 = (M - \Delta)/2$ 
# We will now compare the exact solution stated above
# to the numerical solution from the Euler method

```

```

k <- 1
M <- 4
h <- 3

```

```

Delta <- sqrt(M^2 - 4 * h/k)
M1 <- (M + Delta)/2
M2 <- (M - Delta)/2

```

```

exact <- function (P0, col)
{
  t <- seq.int(0, 5, length.out = 101)
  P <- (M1 * (P0 - M2) - M2 * (P0 - M1) * exp(-k * Delta * t)) /
    ((P0 - M2) - (P0 - M1) * exp(-k * Delta * t))
  graphics::lines(t, P, col = col, lwd = 2)
  invisible()
}
euler <- function (P0, col)
{
  t <- seq.int(0, 5, 0.5)
  initialConditions <- c(P = P0)
  fun <- function(t, P) k * P * (M - P) - h
  P <- essentials::EulerMethod(t, initialConditions, fun)$P
  graphics::lines(t, P, col = col, lwd = 2)
  invisible()
}

```

```

P0 <- c(1.5, 2, 3.5) # different values of initial population to plot
exact_colours <- c("red", "green", "blue")
euler_colours <- c("maroon4", "darkgreen", "navy")

```

```

graphics::par(mar = c(5.1, 4.1, 0.4, 0.4))
graphics::plot(
  xlim = c(0, 5), ylim = c(1.5, 3.5),
  panel.first = grid(col = "gray69"),
  x = NA_real_, y = NA_real_,
  bty = "n",

```

```

        xlab = "time", ylab = "population"
    )
    graphics::title(
        sub = bquote(list(k == .(k), M == .(M), h == .(h))),
        adj = 1
    )
    for (i in seq_along(P0)) {
        exact(P0[i], exact_colours[i])
        euler(P0[i], euler_colours[i])
    }
    graphics::legend("bottomright",
        legend = as.expression(essentials::plapply(
            list(
                rep(c("Exact", "Euler"), each = length(P0)),
                rep(P0, 2)
            ),
            function(name, P0) bquote(list(. (name), P[0] == .(P0)))
        )),
        fill = c(exact_colours, euler_colours),
        ncol = 2,
        bty = "n")

```

---

## setReadWriteArgsMethod

*Create and Save a Method for Reading/Writing Command-Line Arguments from/to a File*

---

### Description

Create a method for reading/writing command-line arguments from/to a file, typically by differing file extension.

### Usage

```

has.ext(file, fileext, compression = FALSE, fixed = FALSE,
        ignore.case = TRUE)

scan2(...)

format4scan(x, sep = " ", quote = "\"'", comment.char = "",
            allowEscapes = FALSE, nlines.between.comment.and.args = 0,
            nlines.between.args = 2)

setReadWriteArgsMethod(name, condition, read, write, sealed = FALSE)

```

### Arguments

|             |   |
|-------------|---|
| file        | character vector, the files to test for a file extension.   |
| fileext     | character string, the file extension to be matched. Should contain a <a href="#">regular expression</a> or character string for fixed = TRUE. |
| compression | logical. Should compressed files be included in the pattern matching?   |

|                                 |   |
|---------------------------------|---|
| fixed                           | logical. If TRUE, pattern is a string to be matched as is. Unlike <code>grep()</code> , <code>fixed = TRUE</code> and <code>ignore.case = TRUE</code> may be used together.   |
| ignore.case                     | logical. Should the case of file be ignored when matching against fileext?  |
| ...                             | arguments passed to <code>scan</code> .   |
| sep                             | Empty character string, NULL, or a character string containing just one single-byte character.  |
| quote                           | Character string or NULL, the set of quoting characters.  |
| comment.char                    | character string containing a single character or an empty string. Use "" to turn off the adding comments.  |
| allowEscapes                    | logical. Should C-style escapes be processed?   |
| x                               | any R object. If a list, each element will be turned into a character vector (in the same way as <code>withArgs</code> converts its arguments), otherwise will be turned into a character vector (same as <code>withArgs</code> ). The arguments to be written. |
| nlines.between.comment.and.args | a non-negative integer specifying the number of empty lines between each set of comments and arguments.   |
| nlines.between.args             | a non-negative integer specifying the number of empty lines between each set of arguments. Only used for a list x.  |
| name                            | A character string naming the method.   |
| condition                       | a function accepting a single argument file (the file in which to read/write). Should return TRUE if the file is appropriate for reading/writing with this method, typically by examining the file extension.   |
| read                            | a function accepting a single argument file. Should read the arguments from file.   |
| write                           | a function accepting arguments x, comments, nlines.between.comment.and.args, nlines.between.args. Should turn x into a character vector of arguments to write to a file. This function does <i>NOT</i> have to make use of the other arguments.                 |
| sealed                          | sealed prevents the method being redefined.   |

## Examples

```
# suppose you wanted to define your own method for
# reading/writing command-line arguments to a file. we'll
# say the file extension will be ".myargs". with this, we
# start by making 'condition'
condition <- function(file) {
  essentials::has.ext(file, ".myargs",
    compression = TRUE, fixed = TRUE)
}

# next, we will make a reading function. this will typically
# be some variation of 'scan2', but feel free to use
# anything else that works. for this example, we'll use
# "-" as the delimiter, "\"" as the quoting character, and
# "/" as the comment character
read <- function(file) {
  essentials::scan2(file = file, sep = "-",
    quote = "\"", comment.char = "/")
}
```



```

}

# next, we will make a writing function. this will typically
# be some variation of 'format4scan', but feel free to use
# anything else that works
write <- function(x, comments = TRUE,
  nlines.between.comment.and.args = 0,
  nlines.between.args = 2) {
  essentials::format4scan(x, sep = "-", quote = "`",
    comment.char = if (comments) "/" else "",
    nlines.between.comment.and.args = nlines.between.comment.and.args,
    nlines.between.args = nlines.between.args)
}

# now, combine it all together
essentials::setReadWriteArgsMethod(
  name      = "myargs",
  condition = condition,
  read      = read,
  write     = write
)

# try writing arguments with this new format
x <- letters
comment(x) <- "testing comments"
essentials::writeArgs(x, "", name = "myargs")

# confirm that writing and reading returns the same set of
# arguments
FILE <- essentials::writeArgs(x, fileext = ".myargs", at = FALSE)
y <- essentials::readArgs(FILE)
stopifnot(length(x) == length(y), x == y)

```

---

shEncode

*Quote Strings for Use in OS Shells*


---

## Description

Quote a string to be passed to an operating system shell.

## Usage

```

shEncode(string, type = NULL, unix.type = NULL, windows.type = "Rscript")

commandEncode(string, type = NULL, unix.type = NULL, windows.type = "Rscript")

commandQuote(string, type = NULL, unix.type = NULL, windows.type = "Rscript")

```

## Arguments

|                           |   |
|---------------------------|---|
| <code>string</code>       | a character vector, usually of length one.  |
| <code>type</code>         | character string: the type of quoting. Partial matching is supported.                   |
| <code>unix.type</code>    | character string: the type of quoting under Unix-alikes when <code>type</code> is NULL. |
| <code>windows.type</code> | character string: the type of quoting in Windows when <code>type</code> is NULL.        |

## Details

The choices for quoting are "sh", "perl", "python", "R", "R CMD", "Rcmd", "Rgui", "Rscript", and "Rterm". The default for quoting under Unix-alikes is "sh", while in Windows it is "Rscript".

From my testing with [shQuote](#), it seems like it fails in a few edge cases that `shEncode` can handle correctly.

Those few cases that `shQuote` handles incorrectly are:

- under Unix-alikes with `type = "csh"` and `string` contains a dollar sign or grave accent and `string` has a trailing single quote, `string` loses its trailing single quote.
- in Windows with `type = "cmd"` and `string` contains a double quote preceded by at least one backslash, the preceding backslashes aren't escaped. Also, when `string` contains a set of trailing backslashes, they aren't escaped.

## Value

character vector, the same length as `string`, with the attributes of `string` (after possible coercion to character), excluding class.

## See Also

[Rscript](#), [python](#)

## Examples

```
fun <- function(string) {
  cat(c(
    "string  ", string, "\n",
    "sh      ", essentials::shEncode(string, type = "sh"    ), "\n",
    "Rscript ", essentials::shEncode(string, type = "Rscript"), "\n",
    "R       ", essentials::shEncode(string, type = "R"     ), "\n"
  ), sep = "")
}

fun("abc$def`gh`i\\j")

fun("testing \\\"this\\\"")

fun("\"testing\" $this$ 'out'")

## Not run:
essentials:::.system(paste(c(
  "perl",
  "-e",
  essentials::shEncode(r"{print \"test \\\"this\\\" out\\n\";}",
    windows.type = "perl")
```

```

), collapse = " ")

## End(Not run)

## Not run:
essentials::system(paste(c(
  "python",
  "-c",
  essentials::shEncode(r"{print(\"test \\\"this\\\" out\")}"),
  windows.type = "python")
), collapse = " ")

## End(Not run)

```

shPrompt

*Replicate the Command-Line / Terminal Prompts Seen on a Few Common Shells*

## Description

Get the prompt seen at Windows cmd, Windows Powershell, Bash, and macOS Bash (bash 3.0). The function will return whichever is most appropriate, unless over-ridden by its argument

## Usage

```
shPrompt(type = NULL)
```

## Arguments

type                      NULL or a character string; the shell prompt to return.

## Details

The choices of type are "windows", "cmd", "powershell", "macOS", "bash", "ubuntu", and "unix", with case insensitive partial matching.

If type is a character string, it is matched against the above choices. If type is NULL or does not match one of the above choices, type is pulled from option 'essentials::shPrompt(type)'. If this option is NULL or does not match one of the above choices, type is pulled from environment variable R\_ESSENTIALS\_SH\_PROMPT\_TYPE. If this environment variable is unset or does not match one of the above choices, type is determined by `.Platform$OS.type` and `capabilities("aqua")`.

## Value

character string.

## Examples

```

cat(
  essentials::shPrompt("cmd"           ),
  essentials::shPrompt("powershell"),
  essentials::shPrompt("macOS"        ),
  essentials::shPrompt("bash"         ),
  sep = "\n"
)

```

---

**strip***Remove Leading and Trailing White Spaces*

---

### Description

strip removes leading and trailing white space from each element of a character vector. White space characters include tab, newline, vertical tab, form feed, carriage return, space and possible other locale-dependent characters.

### Usage

```
strip(x)
```

### Arguments

**x** a character vector, or an object which can be coerced by `as.character` to a character vector. [Long vectors](#) are supported.

### Value

strip returns a character vector of the same length and with the same attributes as `x` (after possible coercion to character). Elements of character vectors `x` which are not substituted will be returned unchanged (including any declared encoding).

### Examples

```
x <- c(
  " the quick brown fox jumps over a lazy dog ",
  " the quick brown fox jumps over a lazy dog\t\n"
)
strip(x) # the leading and trailing tab, newline, and space are removed

## x is intended to be in encoding latin1
x <- "fa\xE7ile"
Encoding(x) <- "latin1"
y <- strip(x)

## since 'x' has no leading or trailing white space, 'strip(x)' retains the
## encoding of 'x'
Encoding(y)
y
```

toProv

---

*Convert the Provinces and Territories of Canada to their Names or Postal Abbreviations*


---

## Description

This is mostly intended to solve issues where the names or postal abbreviations may be written incorrectly (in one of the alternate forms listed below).

## Usage

toProv(x)

toProv2(x)

toProvince(x)

## Arguments

x                      character vector. Province and territory names or postal abbreviations.

## Details

The following are the names and postal abbreviations of each province and territory, as well as their accepted alternate forms.

Ontario, ON Ont., O, ØN

Quebec, QC Québec, Province du Québec, Province Québec, Que., Qc, P.Q., PQ, QU, QB

Nova Scotia, NS Nouvelle-Écosse, N.S., N.-É.

New Brunswick, NB Nouveau-Brunswick, N.B., N.-B.

Manitoba, MB Man.

British Columbia, BC Colombie-Britannique, B.C., C.-B.

Prince Edward Island, PE P.E.I., Î.-P.-É., PEI, Île du Prince-Édouard, Île Prince-Édouard

Saskatchewan, SK Sask.

Alberta, AB Alta., Alb. AL

Newfoundland and Labrador, NL Newfoundland &amp; Labrador, Newfoundland, Terre-Neuve-et-Labrador, Terre-Neuve &amp; Labrador, Terre-Neuve, N.L., T.-N.-L., Nfld., T.-N., NF, LB

Northwest Territories, NT Territoires du Nord-Ouest, Territoires Nord-Ouest, N.W.T., T.N.-O.

Yukon, YT Yukon Territory, Yuk., Yn, YK

Nunavut, NU Nvt.

Additionally, the name and postal abbreviation for Canada are Canada and CA. These will rarely be used, but are available if need be. x will be partially matched with the above names, postal abbreviations, and alternate forms (ignoring whitespace and character case).

## Value

character vector of the same length and with the same attributes as x (after possible coercion to character) besides class.

for toProv2, elements of toProv equal to "NL" will return as "NF".

## Examples

```
x <- c(
  "Ontario"                , "ON", "0N",

  # alternate forms
  "Quebec"                 , "QC", "Qu\u{00E9}bec",

  # case insensitive
  "Nova Scotia"           , "NS", "nova scotia",
  "New Brunswick"        , "NB",
  "Manitoba"              , "MB",
  "British Columbia"     , "BC",
  "Prince Edward Island" , "PE", "PEI",

  # partial matching
  "Saskatchewan"         , "SK", "Sask",
  "Alberta"              , "AB", "AL",
  "Newfoundland and Labrador", "NL", "Newfoundland & Labrador", "NF",
  "Northwest Territories", "NT",
  "Yukon"                , "YT", "Yukon Territory", "YK",
  "Nunavut"              , "NU"
)
cbind(Original = x, `Postal Abbr` = toProv(x), Name = toProvince(x))
```

tryExcept

*Condition Handling and Recovery*

## Description

tryExcept provides a mechanism for handling unusual conditions, including errors and warnings.

## Usage

```
tryExcept(expr, ..., finally)
```

## Arguments

|         |  |
|---------|--|
| expr    | expression to be evaluated.                                      |
| ...     | handlers established for the duration of the evaluation of expr. |
| finally | expression to be evaluated before returning or exiting.          |

## Details

If finally is missing or a simple expression, tryExcept and [tryCatch](#) will behave the same.

However, if finally is a *compound expression*, usually of the form { expr1 ; expr2 }, tryExcept will split the compound expression into its elements and put each within their own [on.exit](#), whereas [tryCatch](#) would put the whole compound expression into one [on.exit](#). This means that all elements of finally in tryExcept will be run even if one signals an error, but in [tryCatch](#), finally will run up until the first error.

**Examples**

```
# the following example won't work using 'utils::example' because it signals
# errors, so copy and paste this code into the R Console
```

```
## Not run:
```

```
tryCatch({
  stop("error in 'expr'")
}, finally = {
  stop("tryCatch will not reach the second expression in 'finally'")
  stop("but tryExcept will")
})
```

```
essentials::tryExcept({
  stop("error in 'expr'")
}, finally = {
  stop("tryCatch will not reach the second expression in 'finally'")
  stop("but tryExcept will")
})
```

```
essentials::tryExcept({
  stop("error in 'expr'")
}, finally = {
  cat("this environment = "); print(environment())
  stop("err1")
  stop("err2")
  print(5 + 6)
  stop("err3")
})
```

```
# just checking that the arguments aren't evaluated in the wrong frame
# (why would they be though??)
```

```
expr
finally
```

```
# testing that code with a source reference will be evaluated properly
print(function(x) {
  x # testing comments only appearing in source reference
})
```

```
stop("err4")
print(6 + 7)
```

```
})
```

```
essentials::tryExcept({
  cat("this one should behave the same as tryCatch",
      "because 'finally' is not a compound expression",
      "(is not of class \"{\")",
      sep = "\n")
})
```

```

    stop("error in 'expr'")
  }, finally =

  # checking again that 'finally' is evaluated in the correct environment
  cat("this environment =", utils::capture.output(environment()), "\n"))

essentials::tryExcept({

  cat("Here is a situation in which you might actually use this.",
      "Suppose you're changing a bunch of settings and options that you want",
      "to reset when 'tryExcept' finishes. For example:",
      "* changing the working directory",
      "* changing options stored in `options()`",
      "* changing graphical parameters stored in `graphics::par()`",
      "* shutting down a graphics device with `dev.off()`",
      "* changing the current graphics device with `dev.set()`",
      "* changing the state of `grDevices::devAskNewPage()`",
      "* deleting a temporary file created with `tempfile()`",
      "   or downloaded with `utils::download.file()`",
      "* closing a connection",
      "* changing the random number generator state",
      "   with `RNGkind()` or `set.seed()`",
      "* any other type of cleaning process",
      sep = "\n")
  owd <- getwd()
  oopt <- options(max.print = 10, digits = 17)
  odev <- grDevices::dev.cur()
  if (names(odev) != "null device")
    oldask <- grDevices::devAskNewPage(ask = FALSE)
  FILE <- tempfile()
  setwd(dirname(FILE))
  con <- file(FILE, "w")
  if (has.Random.seed <- exists(".Random.seed", envir = globalenv(), inherits = FALSE))
    oldSeed <- get(".Random.seed", envir = globalenv(), inherits = FALSE)
  else oldRNG <- RNGkind()
  RNGkind("default", "default", "default")
  set.seed(1)

  cat("\nYou've setup your settings and options above",
      "so now we do something with it", sep = "\n")

  cat("\nOnly 10 of these will print\n")
  print(1:100)

  cat("\nWe're plotting an image which we will remove afterwards\n")
  plot(1:10)
  cdev <- grDevices::dev.cur()
  Sys.sleep(2)

}, finally = {

```



```

cat("\nNow clean up the everything, but unlike 'tryCatch', run all",
    "cleaning steps even if one signals an error", sep = "\n")

grDevices::dev.off(cdev)
close(con)
file.remove(FILE)

if (has.Random.seed)
  assign(".Random.seed", oldSeed, envir = globalenv(), inherits = FALSE)
else RNGkind(oldRNG[1L], oldRNG[2L], oldRNG[3L])

if (names(odev) != "null device") {
  grDevices::dev.set(oldask)
  grDevices::devAskNewPage(oldask)
}

setwd(owd)
options(oopt)
})

## End(Not run)

```

withArgs

*Source R Code, Providing Arguments to the Script*

## Description

Evaluate R code from a named file or URL or connection while providing arguments. This would be in the circumstance that you want to run a script and provide command-line arguments, but want the objects to appear in your environment.

## Usage

```
withArgs(expr, ...)
```

## Arguments

|      |  |
|------|--|
| expr | a call to <a href="#">source</a> , <a href="#">sys.source</a> , <a href="#">debugSource</a> , or <a href="#">testthat::source_file</a> . |
| ...  | the arguments provided to the script. See section <b>Details</b> .   |

## Details

... is first put into a list, and then each non-list element is converted to character. They are converted as follows:

**Factors** (class "factor") using [as.character.factor](#)

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

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

**Raw Bytes** (class "raw") using `sprintf("0x%02x", )` (can easily convert back to raw with `as.raw()` or `as.vector(,"raw")`)

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

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

Consider that it may be better to use `Rscript` combined with `save` and `load` or `saveRDS` and `readRDS`.

Also consider that what you want is a function, and not a script. If you're already at the **R** level, it is easier and more flexible to source a script that creates a function, and then use that function. This only applies if the script you are sourcing will never be run from the command-line, or at least will never be run on its own (it will only be used in the context of other scripts, it will *NEVER* be used as a stand alone script).

## Value

the value returned by evaluating `expr`, see the help pages of the above source functions.

## Examples

```
essentials:::write.code(file = FILE <- tempfile(), {
  withAutoprint({

    this.path::this.path()
    essentials::Args()

  }, verbose = FALSE)
})

# wrap your source call with a call to `withArgs`
essentials::withArgs(
  source(FILE, local = TRUE, verbose = FALSE),
  letters, pi, exp(1)
)
essentials::withArgs(
  sys.source(FILE, environment()),
  letters, pi + 1i * exp(1)
)

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

---

wrapper

---

*Wrapper Functions*

---

**Description**

Create a call that can be substituted into a wrapper function.

**Usage**

```
wrapper(fun, defaults = NULL, with.pkg = TRUE)
```

**Arguments**

|                       |  |
|-----------------------|--|
| <code>fun</code>      | a character string, symbol or call.  |
| <code>defaults</code> | a list of default arguments for the call, or <code>NULL</code> .                             |
| <code>with.pkg</code> | logical; if the function is from a namespace, should the namespace name be used in the call? |

**Details**

A wrapper function is a function whose purpose is to call another function. Wrapper functions are useful for hiding details of a function's implementation.

**Value**

A call.

**Examples**

```
# I don't particularly like that the function data.frame has
# the formal argument "check.names" default to TRUE. Here, we
# will use 'wrapper' to make a wrapper for data.frame that has
# "check.names" set to FALSE

# we want the function body to look like this
wrapper(data.frame)

## make the function with the appropriate function body
data.frame2 <- function() NULL
body(data.frame2) <- wrapper(data.frame)

## add the function formals, changing "check.names" from TRUE to FALSE
formals(data.frame2) <- formals(data.frame)
formals(data.frame2)$check.names <- FALSE

print(data.frame2)
```

# Index

- \* **classes**
  - numbers-class, 38
- \* **package**
  - essentials-package, 2
- .Platform, 59
- .mapply, 34
- .plapply, 3, 41, 42
- .psapply (.plapply), 3
- .pvapply (.plapply), 3
- [[, 4, 41
- %row.in% (rowmatch), 50
- %row.in% (rowmatch), 50
- %until% (do.while), 20
- %while% (do.while), 20
  
- add.argument, 5, 7, 10, 11
- add.argument, ArgumentParser-method
  - (add.argument), 5
- add.argument, essentials\_ArgumentParser-method
  - (add.argument), 5
- add.help, 6, 10
- add.help, ArgumentParser-method
  - (add.help), 6
- add.help, essentials\_ArgumentParser-method
  - (add.help), 6
- add.skip (add.help), 6
- add.skip, ArgumentParser-method
  - (add.help), 6
- add.skip, essentials\_ArgumentParser-method
  - (add.help), 6
- add.subparsers, 7, 11
- add.subparsers, ArgumentParser-method
  - (add.subparsers), 7
- add.subparsers, essentials\_ArgumentParser-method
  - (add.subparsers), 7
- add.version (add.help), 6
- add.version, ArgumentParser-method
  - (add.help), 6
- add.version, essentials\_ArgumentParser-method
  - (add.help), 6
- Args, 8
- ArgumentParser, 5, 7, 9, 19
- ArgumentParser-class (ArgumentParser), 9
- Arithmetic, 29
- array, 3, 40, 41
- as.character, 66
- as.character.factor, 65
- as.colorRampPalette, 11
- as.hex.code (color.with.alpha), 16
- as.list, 4, 41
- as.numbers, 38
- as.numbers (numbers), 36
- as.raw, 66
- as.scalar, 12
- as.vector, 37, 66
- ASCII, 13
- aslength1, 14
- asWindowsbasename, 15
  
- baseenv(), 31
- basename, 10, 39
- break, 21
- call, 31
- capabilities, 59
- coerce, ANY, numbers-method
  - (numbers-class), 38
- col2rgb, 16
- color.with.alpha, 16
- colorRamp, 11
- colorRampPalette, 11
- commandArgs, 9, 38
- commandEncode (shEncode), 57
- commandQuote, 45
- commandQuote (shEncode), 57
- Commands, 17
- comments, 46
- complex, 38
- data.frame, 34
- decrement (progressBar), 42
- dedent, 18
- delayedAssign, 19
- delayedAssign2, 19
- dgev (GeneralizedExtremeValue), 25
- dim, 29, 41
- dimnames, 29
- dir2 (list.files2), 33

- do.call, [41](#)
- do.until (do.while), [20](#)
- do.while, [20](#)
- double, [37](#)
- environment, [31](#)
- Environment Variables, [23](#)
- envvar (envvars), [22](#)
- envvars, [22](#)
- essentials (essentials-package), [2](#)
- essentials-package, [2](#)
- essentials\_ArgumentParser-class  
(ArgumentParser), [9](#)
- EulerMethod (Runge-Kutta Methods), [51](#)
- expression, [31](#)
- extends, [34](#)
- file.open, [24](#)
- file.path, [39](#)
- flat.list, [24](#)
- format4scan (setReadWriteArgsMethod), [55](#)
- formula, [31](#)
- GeneralizedExtremeValue, [25](#)
- getEnvvar (envvars), [22](#)
- getProgress (progressBar), [42](#)
- gg.colors (hcl.colors2), [27](#)
- globalenv, [43](#)
- graphics::legend, [31](#)
- grep, [56](#)
- gzfile, [47](#)
- has.ext (setReadWriteArgsMethod), [55](#)
- hcl.colors, [28](#)
- hcl.colors2, [27](#)
- hcl.pals, [28](#)
- hypot, [28](#)
- IDW (InverseDistanceWeighting), [30](#)
- ImprovedEulerMethod (Runge-Kutta  
Methods), [51](#)
- increment (progressBar), [42](#)
- inferno.colors (hcl.colors2), [27](#)
- inherits, [34](#)
- InverseDistanceWeighting, [30](#)
- is.numbers (numbers), [36](#)
- is.vector, [15](#)
- lapply, [40](#), [41](#)
- legend.dimensions, [31](#)
- length, [4](#), [41](#)
- lengths, [41](#)
- list.files, [33](#)
- list.files2, [33](#)
- listify, [34](#)
- load, [66](#)
- Long vectors, [60](#)
- mapply, [34](#), [41](#)
- match, [50](#)
- match.fun, [3](#), [40](#)
- Missing, [35](#)
- mode, [37](#)
- names, [3](#), [4](#), [29](#), [40](#), [41](#)
- new, [34](#)
- new.env, [43](#)
- next, [21](#)
- normalizeAgainst, [36](#)
- normalized, [36](#)
- normalizePath, [36](#), [39](#)
- numbers, [36](#)
- numbers-class, [38](#)
- numeric, [37](#), [38](#)
- oldClass, [34](#)
- on.exit, [62](#)
- options, [23](#)
- par, [14](#)
- parse.args, [5](#), [38](#)
- parse.args, ArgumentParser-method  
(parse.args), [38](#)
- parse.args, essentials\_ArgumentParser-method  
(parse.args), [38](#)
- path.contract, [39](#)
- path.expand, [39](#)
- pgev (GeneralizedExtremeValue), [25](#)
- phypot (hypot), [28](#)
- plapply, [4](#), [40](#)
- plasma.colors (hcl.colors2), [27](#)
- progressBar, [42](#)
- psapply (plapply), [40](#)
- pseudoglobalenv, [43](#)
- pvapply (plapply), [40](#)
- python, [43](#), [58](#)
- qgev (GeneralizedExtremeValue), [25](#)
- Quotes, [46](#)
- R, [44](#)
- rainbow, [11](#)
- rapplly, [25](#)
- Rcmd (R), [44](#)
- readArgs, [45](#)
- readRDS, [66](#)
- reference, [10](#)
- regex, [6](#)

- regular expression, 55
- repeat, 21
- return, 21
- rgev (GeneralizedExtremeValue), 25
- RK1 (Runge-Kutta Methods), 51
- RK2 (Runge-Kutta Methods), 51
- RK4 (Runge-Kutta Methods), 51
- row.match (rowmatch), 50
- row.match, data.frame, data.frame-method (rowmatch), 50
- row.match.data.frame (rowmatch), 50
- row.names, 51
- rowmatch, 50
- rownames, 51
- Rscript, 58, 66
- Rscript (R), 44
- Rterm (R), 44
- Runge-Kutta Methods, 51
- RungeKuttaMethod (Runge-Kutta Methods), 51
- sapply, 40, 41
- save, 66
- saveRDS, 66
- scalar (as.scalar), 12
- scan, 47, 56
- scan2 (setReadWriteArgsMethod), 55
- setProgress (progressBar), 42
- setProgressBarMethod (progressBar), 42
- setReadWriteArgsMethod, 55
- shEncode, 57
- show.colors (hcl.colors2), 27
- shPrompt, 59
- shQuote, 58
- simplify2array, 41
- source, 43, 65
- sprintf, 66
- stdout, 46
- strip, 60
- structure, 34
- sys.call, 31
- sys.source, 65
- system, 44, 45
- vapply, 40, 41
- viridis.colors, 11
- viridis.colors (hcl.colors2), 27
- while, 21
- withArgs, 9, 44–46, 56, 65
- wrapped, 5, 7, 8, 10
- wrapper, 67
- writeArgs (readArgs), 45
- sapply, 40, 41
- save, 66
- saveRDS, 66
- scalar (as.scalar), 12
- scan, 47, 56
- scan2 (setReadWriteArgsMethod), 55
- setProgress (progressBar), 42
- setProgressBarMethod (progressBar), 42
- setReadWriteArgsMethod, 55
- shEncode, 57
- show.colors (hcl.colors2), 27
- shPrompt, 59
- shQuote, 58
- simplify2array, 41
- source, 43, 65
- sprintf, 66
- stdout, 46
- strip, 60
- structure, 34
- sys.call, 31
- sys.source, 65
- system, 44, 45
- testthat::source\_file, 65
- tkProgressBar, 43
- toProv, 61
- toProv2 (toProv), 61
- toProvince (toProv), 61
- tryCatch, 62
- tryExcept, 62
- txtProgressBar, 43