Package 'iotools'

May 8, 2026

Version 0.4-0

Title I/O Tools for Streaming

Author Simon Urbanek [aut, cre] (https://urbanek.nz, ORCID:
<https://orcid.org/0000-0003-2297-1732>),
Taylor Arnold [aut]

Maintainer Simon Urbanek <Simon.Urbanek@r-project.org>

Depends R (>= 2.9.0)

Imports methods, parallel, utils

Description Basic I/O tools for streaming and data parsing.

License GPL-2 | GPL-3

URL https://www.rforge.net/iotools

NeedsCompilation yes

Repository CRAN

Date/Publication 2026-02-19 06:20:03 UTC

Contents

`.default.formatter` *Default formatter, coorisponding to the as.output functions*

Description

This function provides the default formatter for the iotools package; it assumes that the key is seperated from the rest of the row by a tab character, and the elements of the row are seperated by the pipe ("|") character. Vector and matrix objects returned from the output via `as.output`.

Usage

```
.default.formatter(x)
```

Arguments

`x`	character vector (each element is treated as a row) or a raw vector (LF characters `'\n'` separate rows) to split

Value

Either a character matrix with a row for each element in the input, or a character vector with an element for each element in the input. The latter occurs when only one column (not counting the key) is detected in the input. The keys are stored as rownames or names, respectively.

Author(s)

Simon Urbanek

See Also

`mstrsplit`

Examples

```
c <- c("A\tB|C|D", "A\tB|B|B", "B\tA|C|E")
.default.formatter(c)

c <- c("A\tD", "A\tB", "B\tA")
.default.formatter(c)
```

`as.output` *Character Output*

Description

Create objects of class `output`.

Usage

```
as.output(x, ...)
```

Arguments

`x`	object to be converted to an instance of `output`.
`...`	optional arguments to be passed to implementing methods of `as.output`. Most methods support the following arguments: `sep` string, column/value separator, `nsep` string, key separator, `keys` either a logical (if `FALSE` names/row names are suppressed) or a character vector with overriding keys. The default for `keys` typically varies by class or is auto-detected (e.g., named vectors user names as keys, `data.frames` use row names if they are non-automatic etc.). All methods also support `con` argument which pushes the output into a connection instead of generating an output object - so `as.output(x, con=...)` is thus not a coersion but used only for its side-effect. Note that `con` also supports special values `iotools.stdout`, `iotools.stderr` and `iotools.fd(fd)` which write directly into the corresponding streams instead of using theconnection API.

Details

`as.output` is generic, and methods can be written to support new classes. The output is meant to be a raw vector suitable for writing to the disk or sending over a connection.

Value

if `con` is set to a connection then the result is `NULL` and the method is used for its side-effect, otherwise the result is a raw vector.

Side note: we cannot create a formal type of `output`, because `writeBin` does `is.vector()` check which doesn't dispatch and prevents anything with a class to be written.

Author(s)

Simon Urbanek

Examples

```
m = matrix(sample(letters), ncol=2)
as.output(m)

df = data.frame(a = sample(letters), b = runif(26), c = sample(state.abb,26))
str(as.output(df))

as.output(df, con=iotools.stdout)
```

---

chunk *Functions for very fast chunk-wise processing*

Description

`chunk.reader` creates a reader that will read from a binary connection in chunks while preserving integrity of lines.

`read.chunk` reads the next chunk using the specified reader.

Usage

```
chunk.reader(source, max.line = 65536L, sep = NULL)
read.chunk(reader, max.size = 33554432L, timeout = Inf)
```

Arguments

`source`	binary connection or character (which is interpreted as file name) specifying the source
`max.line`	maximum length of one line (in byets) - determines the size of the read buffer, default is 64kb
`sep`	optional string: key separator if key-aware chunking is to be used

character is considered a key and subsequent records holding the same key are guaranteed to be

`reader`	reader object as returned by `chunk.reader`
`max.size`	maximum size of the chunk (in bytes), default is 32Mb
`timeout`	numeric, timeout (in seconds) for reads if `source` is a raw file descriptor.

Details

`chunk.reader` is essentially a filter that converts binary connection into chunks that can be subsequently parsed into data while preserving the integrity of input lines. `read.chunk` is used to read the actual chunks. The implementation is very thin to prevert copying of large vectors for best efficiency.

If `sep` is set to a string, it is treated as a single-character separator character. If specified, prefix in the input up to the specified character is treated as a key and subsequent lines with the same key are guaranteed to be processed in the same chunk. Note that this implies that the chunk size is

practically unlimited, since this may force accumulation of multiple chunks to satisfy this condition. Obviously, this increases the processing and memory overhead.

In addition to connections `chunk.reader` supports raw file descriptors (integers of the class `"fileDescriptor"`). In that case the reads are preformed directly by `chunk.reader` and `timeout` can be used to perform non-blocking or timed reads (unix only, not supported on Windows).

Value

`chunk.reader` returns an object that can be used by `read.chunk`. If `source` is a string, it is equivalent to calling `chunk.reader(file(source, "rb"), ...)`.

`read.chunk` returns a raw vector holding the next chunk or `NULL` if timeout was reached. It is deliberate that `read.chunk` does NOT return a character vector since that would reasult in a high performance penalty. Please use the appropriate parser to convert the chunk into data, see `mstrsplit`.

Author(s)

Simon Urbanek

---

chunk.apply *Process input by applying a function to each chunk*

Description

`chunk.apply` processes input in chunks and applies `FUN` to each chunk, collecting the results.

Usage

```
chunk.apply(input, FUN, ..., CH.MERGE = rbind, CH.MAX.SIZE = 33554432,
            CH.PARALLEL=1L, CH.SEQUENTIAL=TRUE, CH.BINARY=FALSE,
            CH.INITIAL=NULL)

chunk.tapply(input, FUN, ..., sep, CH.MERGE = rbind, CH.MAX.SIZE = 33554432)
```

Arguments

`input`	Either a chunk reader or a file name or connection that will be used to create a chunk reader
`FUN`	Function to apply to each chunk
`...`	Additional parameters passed to `FUN`
`sep`	singe character string. For `tapply`, gives separator for the key over which to apply. Each line is split at the first separator, and the preceding value is treated as the key over which to apply the function. If the `input` is a chunk reader, then this value is ignored (can be missing) and the key separator of the chunk reader is always used, otherwise defaults to `"\t"` and the corresponding chunk reader with that key separator is created internally.

CH.MERGE	Function to call to merge results from all chunks. Common values are list to get lapply-like behavior, rbind for table-like output or c for a long vector.
CH.MAX.SIZE	maximal size of each chunk in bytes
CH.PARALLEL	the number of parallel processes to use in the calculation (unix only).
CH.SEQUENTIAL	logical, only relevant for parallel processing. If TRUE then the chunks are guaranteed to be processed in sequential order. If FALSE then the chunks may be processed in any order to gain better performance.
CH.BINARY	logical, if TRUE then CH.MERGE is a binary function used to update the result object for each chunk, effectively acting like the Reduce function. If FALSE then the results from all chunks are accumulated first and then CH.MERGE is called with all chunks as arguments. See below for performance considerations.
CH.INITIAL	Function which will be applied to the first chunk if CH.BINARY=TRUE. If NULL then CH.MERGE(NULL, chunk) is called instead.

Details

Due to the fact that chunk-wise processing is typically used when the input data is too large to fit in memory, there are additional considerations depending on whether the results after applying FUN are itself large or not. If they are not, then the apporach of accumulating them and then applying CH.MERGE on all results at once is typically the most efficient and it is what CH.BINARY=FALSE will do.

However, in some situations where the result are resonably big or the number of chunks is very high, it may be more efficient to update a sort of state based on each arriving chunk instead of collecting all results. This can be achieved by setting CH.BINARY=TRUE in which case the process is equivalent to:

```
res <- CH.INITIAL(FUN(chunk1))
res <- CH.MERGE(res, FUN(chunk2))
res <- CH.MERGE(res, FUN(chunk3))
...
res
```

If CH.INITITAL is NULL then the first line is res <- CH.MERGE(NULL, FUN(chunk1)).

The parameter CH.SEQUENTIAL is only used if parallel processing is requested. It allows the system to process chunks out of order for performace reasons. If it is TRUE then the order of the chunks is respected, but merging can only proceed if the result of the next chunk is avaiable. With CH.SEQUENTIAL=FALSE the workers will continue processing further chunks as they become avaiable, not waiting for the results of the preceding calls. This is more efficient, but the order of the chunks in the result is not deterministic.

Note that if parallel processing is required then all calls to FUN should be considered independent. However, CH.MERGE is always run in the current session and thus is allowed to have side-effects.

chunk.tapply requires that the input is sharded by key, i.e. records with the same key must be adjacent (similar to ctapply). The function FUN is then guaranteed to be called for all values of exactly one key at a time (unlike chunk.apply which always processes the entire chunk which may contain multiple keys).

Value

The result of calling CH.MERGE on all chunk results as arguments (CH.BINARY=FALSE) or result of the last call to binary CH.MERGE.

Note

The input to FUN is the raw chunk, so typically it is advisable to use mstrsplit or similar function as the first step in FUN.

Note

The support for CH.PARALLEL is considered experimental and may change in the future.

Author(s)

Simon Urbanek

Examples

```
## Not run:
## compute quantiles of the first variable for each chunk
## of at most 10kB size
chunk.apply("input.file.txt",
            function(o) {
              m = mstrsplit(o, type='numeric')
              quantile(m[,1], c(0.25, 0.5, 0.75))
            }, CH.MAX.SIZE=1e5)

## End(Not run)
```

---

chunk.map *Map a function over a file by chunks*

Description

A wrapper around the core iotools functions to easily apply a function over chunks of a large file. Results can be either written to a file or returned as an internal list.

Usage

```
chunk.map(input, output = NULL, formatter = .default.formatter,
          FUN, key.sep = NULL, max.line = 65536L,
          max.size = 33554432L, output.sep = ",", output.nsep = "\t",
          output.keys = FALSE, skip = 0L, ...)
```

Arguments

input	an input connection or character vector describing a local file.
output	an optional output connection or character vector describing a local file. If NULL, the results are returned internally as a list.
formatter	a function that takes raw input and produces the input given to FUN
FUN	a user provided function to map over the chunks. The result of FUN is either wrapper in a list item, when output is NULL, or written to the output file using as.output
key.sep	optional key separator given to chunk.reader
max.line	maximum number of lines given to chunk.reader
max.size	maximum size of a block as given to read.chunk
output.sep	single character giving the field separator in the output.
output.nsep	single character giving the key separator in the output.
output.keys	logical. Whether as.output should interpret row names as keys.
skip	integer giving the number of lines to strip off the input before reading. Useful when the input contains a row a column headers
...	additional parameters to pass to FUN

Value

A list of results when output is NULL; otherwise no output is returned.

Author(s)

Taylor Arnold

---

ctapply *Fast tapply() replacement function for contiguous groups*

Description

ctapply is a fast replacement of tapply that assumes contiguous input, i.e. unique values in the index are never separated by any other values. This avoids an expensive split step since both value and the index chunks can be created on the fly. This can make it orders of magnitude faster than the classical lapply(split(), ...) implementation.

Usage

```
ctapply(X, INDEX, FUN, ..., MERGE=c)
```

Arguments

X	an atomic object, typically a vector
INDEX	numeric or character vector of the same length as X
FUN	the function to be applied
...	additional arguments to FUN. They are passed as-is, i.e., without replication or recycling
MERGE	function to merge the resulting vector or NULL if the arguments to such a function are to be returned instead

Details

Note that ctapply supports either integer, real or character vectors as indices (note that factors are integer vectors and thus supported; you do not need to convert character vectors). Unlike tapply it does not take a list of factors - if you want to use a cross-product of factors, create the product first, e.g. using paste(i1, i2, i3, sep='\01') or multiplication - whetever method is convenient for the input types.

ctapply requires the INDEX to contiguous. One (slow) way to achieve that is to use sort or order, but in typical use-cases it is applied to already structured data which is sharded, but does not need to be sorted.

ctapply also supports X to be a matrix in which case it is split row-wise based on INDEX. The number of rows must match the length of INDEX. Note that the indexed matrices behave as if drop=FALSE was used and currently dimnames are only honored if rownames are present.

If the output is multi-dimensional, you probably want to use MERGE=rbind or MERGE=cbind instead of the default.

Note

This function has been moved to the fastmatch package!

Author(s)

Simon Urbanek

See Also

tapply

Examples

```
# contiguous names = LETTERS with ~350k values each
l <- rep(LETTERS, rnorm(length(LETTERS), 350000, 10000))
# random values
i <- rnorm(length(l))

system.time(rt <- tapply(i, l, sum))
system.time(rc <- ctapply(i, l, sum))
## tapply always returns an array so compare the same structure
identical(rt, as.array(rc))
```

```
## ctapply() also works on matrices (unlike tapply)
m <- matrix(c("A","A","B","B","B","C","A","B","C","D","E","F","","X","X","Y","Y","Z"),,3)
ctapply(m, m[,1], identity, MERGE=list)
ctapply(m, m[,1], identity, MERGE=rbind)
m2 <- m[,-1]
rownames(m2) <- m[,1]
colnames(m2) <- c("V1","V2")
ctapply(m2, rownames(m2), identity, MERGE=list)
ctapply(m2, rownames(m2), identity, MERGE=rbind)
```

`dstrfw` *Split fixed width input into a dataframe*

Description

`dstrfw` takes raw or character vector and splits it into a dataframe according to a vector of fixed widths.

Usage

```
dstrfw(x, col_types, widths, nsep = NA, strict=TRUE, skip=0L, nrows=-1L)
```

Arguments

`x`	character vector (each element is treated as a row) or a raw vector (newlines separate rows)
`col_types`	required character vector or a list. A vector of classes to be assumed for the output dataframe. If it is a list, `class(x)[1]` will be used to determine the class of the contained element. It will not be recycled, and must be at least as long as the longest row if `strict` is `TRUE`. Possible values are `"NULL"` (when the column is skipped) one of the six atomic vector types (`'character'`, `'numeric'`, `'logical'`, `'integer'`, `'complex'`, `'raw'`) or `POSIXct`. 'POSIXct' will parse date format in the form "YYYY-MM-DD hh:mm:ss.sss" assuming GMT time zone. The separators between digits can be any non-digit characters and only the date part is mandatory. See also `fasttime::asPOSIXct` for details.
`widths`	a vector of widths of the columns. Must be the same length as `col_types`.
`nsep`	index name separator (single character) or `NA` if no index names are included
`strict`	logical, if `FALSE` then `dstrsplit` will not fail on parsing errors, otherwise input not matching the format (e.g. more columns than expected) will cause an error.
`skip`	integer: the number of lines of the data file to skip before beginning to read data.
`nrows`	integer: the maximum number of rows to read in. Negative and other invalid values are ignored.

Details

If nsep is specified, the output of dstrsplit contains an extra column called 'rowindex' containing the row index. This is used instead of the rownames to allow for duplicated indicies (which are checked for and not allowed in a dataframe, unlike the case with a matrix).

Value

If nsep is specified then all characters up to (but excluding) the occurrence of nsep are treated as the index name. The remaining characters are split using the widths vector into fields (columns). dstrfw will fail with an error if any line does not contain enough characters to fill all expected columns, unless strict is FALSE. Excessive columns are ignored in that case. Lines may contain fewer columns (but not partial ones unless strict is FALSE) in which case they are set to NA.

dstrfw returns a data.frame with as many rows as they are lines in the input and as many columns as there are non-NA values in col_types, plus an additional column if nsep is specified. The colnames (other than the row index) are set to 'V' concatenated with the column number unless col_types is a named vector in which case the names are inherited.

Author(s)

Taylor Arnold and Simon Urbanek

Examples

```
input = c("bear\t22.7horse+3", "pear\t 3.4mouse-3", "dogs\t14.8prime-8")
z = dstrfw(x = input, col_types = c("numeric", "character", "integer"),
      width=c(4L,5L,2L), nsep="\t")
z

# Now without row names (treat seperator as a 1 char width column with type NULL)
z = dstrfw(x = input,
    col_types = c("character", "NULL", "numeric", "character", "integer"),
    width=c(4L,1L,4L,5L,2L))
z
```

---

dstrsplit *Split binary or character input into a dataframe*

Description

dstrsplit takes raw or character vector and splits it into a dataframe according to the separators.

Usage

```
dstrsplit(x, col_types, sep="|", nsep=NA, strict=TRUE, skip=0L, nrows=-1L,
          quote="")
```

Arguments

x	character vector (each element is treated as a row) or a raw vector (newlines separate rows)
col_types	required character vector or a list. A vector of classes to be assumed for the output dataframe. If it is a list, `class(x)[1]` will be used to determine the class of the contained element. It will not be recycled, and must be at least as long as the longest row if `strict` is `TRUE`. Possible values are `"NULL"` (when the column is skipped) one of the six atomic vector types (`'character'`, `'numeric'`, `'logical'`, `'integer'`, `'complex'`, `'raw'`) or `POSIXct`. 'POSIXct' will parse date format in the form "YYYY-MM-DD hh:mm:ss.sss" assuming GMT time zone. The separators between digits can be any non-digit characters and only the date part is mandatory. See also `fasttime::asPOSIXct` for details.
sep	single character: field (column) separator. Set to `NA` for no seperator; in other words, a single column.
nsep	index name separator (single character) or `NA` if no index names are included
strict	logical, if `FALSE` then `dstrsplit` will not fail on parsing errors, otherwise input not matching the format (e.g. more columns than expected) will cause an error.
skip	integer: the number of lines of the data file to skip before beginning to read data.
nrows	integer: the maximum number of rows to read in. Negative and other invalid values are ignored.
quote	the set of quoting characters as a length 1 vector. To disable quoting altogether, use `quote = ""` (the default). Quoting is only considered for columns read as character.

Details

If `nsep` is specified then all characters up to (but excluding) the occurrence of `nsep` are treated as the index name. The remaining characters are split using the `sep` character into fields (columns). `dstrsplit` will fail with an error if any line contains more columns then expected unless `strict` is `FALSE`. Excessive columns are ignored in that case. Lines may contain fewer columns in which case they are set to `NA`.

Note that it is legal to use the same separator for `sep` and `nsep` in which case the first field is treated as a row name and subsequent fields as data columns.

If `nsep` is specified, the output of `dstrsplit` contains an extra column called 'rowindex' containing the row index. This is used instead of the rownames to allow for duplicated indicies (which are checked for and not allowed in a dataframe, unlike the case with a matrix).

Value

`dstrsplit` returns a data.frame with as many rows as they are lines in the input and as many columns as there are non-NULL values in `col_types`, plus an additional column if `nsep` is specified. The colnames (other than the row index) are set to 'V' concatenated with the column number unless `col_types` is a named vector in which case the names are inherited.

Author(s)

Taylor Arnold and Simon Urbanek

Examples

```
input = c("apple\t2|2.7|horse|0d|1|2015-02-05 20:22:57",
          "pear\t7|3e3|bear|e4|1+3i|2015-02-05",
          "pear\te|1.8|bat|77|4.2i|2001-02-05")
z = dstrsplit(x = input,
      col_types = c("integer", "numeric", "character","raw","complex","POSIXct"),
      sep="|", nsep="\t")
lapply(z,class)
z

# Ignoring the third column:
z = dstrsplit(x = input,
      col_types = c("integer", "numeric", "character","raw","complex","POSIXct"),
      sep="|", nsep="\t")
z
```

---

fdrbind *Fast row-binding of lists and data frames*

Description

`fdrbind` lakes a list of data frames or lists and merges them together by rows very much like `rbind` does for its arguments. But unlike `rbind` it specializes on data frames and lists of columns only and performs the merge entriley at C leve which allows it to be much faster than `rbind` at the cost of generality.

Usage

```
fdrbind(list)
```

Arguments

`list`	lists of parts that can be either data frames or lists

Details

All parts are expected to have the same number of columns in the same order. No column name matching is performed, they are merged by position. Also the same column in each part has to be of the same type, no coersion is performed at this point. The first part determines the column names, if any. If the parts contain data frames, their rownames are ignored, only the contents are merged. Attributes are not copied, which is intentional. Probaby the most common implocation is that ff you use factors, they must have all the same levels, otherwise you have to convert factor columns to strings first.

Value

The merged data frame.

Author(s)

Simon Urbanek

See Also

rbind

`idstrsplit` *Create an iterator for splitting binary or character input into a dataframe*

Description

`idstrsplit` takes a binary connection or character vector (which is interpreted as a file name) and splits it into a series of dataframes according to the separator.

Usage

```
idstrsplit(x, col_types, sep="|", nsep=NA, strict=TRUE,
           max.line = 65536L, max.size = 33554432L)
```

Arguments

`x`	character vector (each element is treated as a row) or a raw vector (newlines separate rows)
`col_types`	required character vector or a list. A vector of classes to be assumed for the output dataframe. If it is a list, `class(x)[1]` will be used to determine the class of the contained element. It will not be recycled, and must be at least as long as the longest row if `strict` is `TRUE`. Possible values are `"NULL"` (when the column is skipped) one of the six atomic vector types (`'character'`, `'numeric'`, `'logical'`, `'integer'`, `'complex'`, `'raw'`) or `POSIXct`. 'POSIXct' will parse date format in the form "YYYY-MM-DD hh:mm:ss.sss" assuming GMT time zone. The separators between digits can be any non-digit characters and only the date part is mandatory. See also `fasttime::asPOSIXct` for details.
`sep`	single character: field (column) separator. Set to `NA` for no seperator; in other words, a single column.
`nsep`	index name separator (single character) or `NA` if no index names are included
`strict`	logical, if `FALSE` then `dstrsplit` will not fail on parsing errors, otherwise input not matching the format (e.g. more columns than expected) will cause an error.
`max.line`	maximum length of one line (in byets) - determines the size of the read buffer, default is 64kb
`max.size`	maximum size of the chunk (in bytes), default is 32Mb

Details

If nsep is specified then all characters up to (but excluding) the occurrence of nsep are treated as the index name. The remaining characters are split using the sep character into fields (columns). dstrsplit will fail with an error if any line contains more columns then expected unless strict is FALSE. Excessive columns are ignored in that case. Lines may contain fewer columns in which case they are set to NA.

Note that it is legal to use the same separator for sep and nsep in which case the first field is treated as a row name and subsequent fields as data columns.

If nsep is specified, the output of dstrsplit contains an extra column called 'rowindex' containing the row index. This is used instead of the rownames to allow for duplicated indicies (which are checked for and not allowed in a dataframe, unlike the case with a matrix).

Value

idstrsplit returns an iterator (closure). When nextElem is called on the iterator a data.frame is returned with as many rows as they are lines in the input and as many columns as there are non-NULL values in col_types, plus an additional column if nsep is specified. The colnames (other than the row index) are set to 'V' concatenated with the column number unless col_types is a named vector in which case the names are inherited.

Author(s)

Michael Kane

Examples

```
col_names <- names(iris)
write.csv(iris, file="iris.csv", row.names=FALSE)
it <- idstrsplit("iris.csv", col_types=c(rep("numeric", 4), "character"),
                 sep=",")
# Get the elements
iris_read <- it$nextElem()[-1,]
# or with the iterators package
# nextElem(it)
names(iris_read) <- col_names
print(head(iris_read))

## remove iterator, connections and files
rm("it")
gc(FALSE)
unlink("iris.csv")
```

---

imstrsplit *Create an iterator for splitting binary or character input into a matrix*

Description

imstrsplit takes a binary connection or character vector (which is interpreted as a file name) and splits it into a character matrix according to the separator.

Usage

```
imstrsplit(x, sep="|", nsep=NA, strict=TRUE, ncol = NA,
          type=c("character", "numeric", "logical", "integer",  "complex",
                 "raw"), max.line = 65536L, max.size = 33554432L)
```

Arguments

x	character vector (each element is treated as a row) or a raw vector (LF characters '\n' separate rows) to split
sep	single character: field (column) separator. Set to NA for no seperator; in other words, a single column.
nsep	row name separator (single character) or NA if no row names are included
strict	logical, if FALSE then mstrsplit will not fail on parsing errors, otherwise input not matching the format (e.g. more columns than expected) will cause an error.
ncol	number of columns to expect. If NA then the number of columns is guessed from the first line.
type	a character string representing one of the 6 atomic types: 'character', 'numeric', 'logical', 'integer', 'complex', or 'raw'. The output matrix will use this as its storage mode and the input will be parsed directly into this format without using intermediate strings.
max.line	maximum length of one line (in byets) - determines the size of the read buffer, default is 64kb
max.size	maximum size of the chunk (in bytes), default is 32Mb

Details

If the input is a raw vector, then it is interpreted as ASCII/UTF-8 content with LF ('\n') characters separating lines. If the input is a character vector then each element is treated as a line.

If nsep is specified then all characters up to (but excluding) the occurrence of nsep are treated as the row name. The remaining characters are split using the sep character into fields (columns). If ncol is NA then the first line of the input determines the number of columns. mstrsplit will fail with an error if any line contains more columns then expected unless strict is FALSE. Excessive columns are ignored in that case. Lines may contain fewer columns in which case they are set to NA.

The processing is geared towards efficiency - no string re-coding is performed and raw input vector is processed directly, avoiding the creation of intermediate string representations.

Note that it is legal to use the same separator for sep and nsep in which case the first field is treated as a row name and subsequent fields as data columns.

Value

A matrix with as many rows as they are lines in the input and as many columns as there are fields in the first line. The storage mode of the matrix will be determined by the input to `type`.

Author(s)

Michael Kane

Examples

```
mm <- model.matrix(~., iris)
f <- file("iris_mm.io", "wb")
writeBin(as.output(mm), f)
close(f)
it <- imstrsplit("iris_mm.io", type="numeric", nsep="\t")
iris_mm <- it$nextElem()
print(head(iris_mm))

## remove iterator, connections and files
rm("it")
gc(FALSE)
unlink("iris_mm.io")
```

`input.file` *Load a file on the disk*

Description

`input.file` efficently reads a file on the disk into R using a formatter function. The function may be `mstrsplit`, `dstrsplit`, `dstrfw`, but can also be a user-defined function.

Usage

```
input.file(file_name, formatter = mstrsplit, ...)
```

Arguments

`file_name`	the input filename as a character string
`formatter`	a function for formatting the input. `mstrsplit` is used by default.
`...`	other arguments passed to the formatter

Value

the return type of the formatter function; by default a character matrix.

Author(s)

Taylor Arnold and Simon Urbanek

line.merge *Merge multiple sources*

Description

Read lines for a collection of sources and merges the results to a single output.

Usage

```
line.merge(sources, target, sep = "|", close = TRUE)
```

Arguments

sources	A list or vector of connections which need to be merged
target	A connection object or a character string giving the output of the merge. If a character string a new file connection will be created with the supplied file name.
sep	string specifying the key delimiter. Only the first character is used. Can be "" if the entire string is to be treated as a key.
close	logical. Should the input to sources be closed by the function.

Value

No explicit value is returned. The function is used purely for its side effects on the sources and target.

Author(s)

Simon Urbanek

mstrsplit *Split binary or character input into a matrix*

Description

mstrsplit takes either raw or character vector and splits it into a character matrix according to the separators.

Usage

```
mstrsplit(x, sep="|", nsep=NA, strict=TRUE, ncol = NA,
        type=c("character", "numeric", "logical", "integer",  "complex", "raw"),
          skip=0L, nrows=-1L, quote="")
```

Arguments

x	character vector (each element is treated as a row) or a raw vector (LF characters '\n' separate rows) to split
sep	single character: field (column) separator. Set to NA for no seperator; in other words, a single column.
nsep	row name separator (single character) or NA if no row names are included
strict	logical, if FALSE then mstrsplit will not fail on parsing errors, otherwise input not matching the format (e.g. more columns than expected) will cause an error.
ncol	number of columns to expect. If NA then the number of columns is guessed from the first line.
type	a character string representing one of the 6 atomic types: 'character', 'numeric', 'logical', 'integer', 'complex', or 'raw'. The output matrix will use this as its storage mode and the input will be parsed directly into this format without using intermediate strings.
skip	integer: the number of lines of the data file to skip before parsing records.
nrows	integer: the maximum number of rows to read in. Negative and other invalid values are ignored, and indiate that the entire input should be processed.
quote	the set of quoting characters as a length 1 vector. To disable quoting altogether, use quote = "" (the default). Quoting is only considered for columns read as character.

Details

If the input is a raw vector, then it is interpreted as ASCII/UTF-8 content with LF ('\n') characters separating lines. If the input is a character vector then each element is treated as a line.

If nsep is specified then all characters up to (but excluding) the occurrence of nsep are treated as the row name. The remaining characters are split using the sep character into fields (columns). If ncol is NA then the first line of the input determines the number of columns. mstrsplit will fail with an error if any line contains more columns then expected unless strict is FALSE. Excessive columns are ignored in that case. Lines may contain fewer columns in which case they are set to NA.

The processing is geared towards efficiency - no string re-coding is performed and raw input vector is processed directly, avoiding the creation of intermediate string representations.

Note that it is legal to use the same separator for sep and nsep in which case the first field is treated as a row name and subsequent fields as data columns.

Value

A matrix with as many rows as they are lines in the input and as many columns as there are fields in the first line. The storage mode of the matrix will be determined by the input to type.

Author(s)

Simon Urbanek

Examples

```
c <- c("A\tB|C|D", "A\tB|B|B", "B\tA|C|E")
m <- mstrsplit(gsub("\t","|",c))
dim(m)
m
m <- mstrsplit(c,, "\t")
rownames(m)
m

## use raw vectors instead
r <- charToRaw(paste(c, collapse="\n"))
mstrsplit(r)
mstrsplit(r, nsep="\t")
```

`output.file` *Write an R object to a file as a character string*

Description

Writes any R object to a file or connection using an output formatter. Useful for pairing with the `input.file` function.

Usage

```
output.file(x, file, formatter.output = NULL)
```

Arguments

`x`	R object to write to the file
`file`	the input filename as a character string or a connection object open for writting.
`formatter.output`	a function for formatting the output. Using null will attempt to find the appropriate method given the class of the input `x`.

Value

invisibly returns the input to `file`.

Author(s)

Taylor Arnold and Simon Urbanek

read.csv.raw *Fast data frame input*

Description

A fast replacement of read.csv and read.delim which pre-loads the data as a raw vector and parses without constructing intermediate strings.

Usage

```
read.csv.raw(file, header=TRUE, sep=",", skip=0L, fileEncoding="",
             colClasses, nrows = -1L, nsep = NA, strict=TRUE,
             nrowsClasses = 25L, quote="'\"")

read.delim.raw(file, header=TRUE, sep="\t", ...)
```

Arguments

file	A connection object or a character string naming a file from which to read data.
header	logical. Does a header row exist for the data.
sep	single character: field (column) separator.
skip	integer. Number of lines to skip in the input, no including the header.
fileEncoding	The name of the encoding to be assumed. Only used when con is a character string naming a file.
colClasses	an optional character vector indicating the column types. A vector of classes to be assumed for the output dataframe. If it is a list, class(x)[1] will be used to determine the class of the contained element. It will not be recycled, and must be at least as long as the longest row if strict is TRUE. Possible values are "NULL" (when the column is skipped) one of the six atomic vector types ('character', 'numeric', 'logical', 'integer', 'complex', 'raw') or POSIXct. 'POSIXct' will parse date format in the form "YYYY-MM-DD hh:mm:ss.sss" assuming GMT time zone. The separators between digits can be any non-digit characters and only the date part is mandatory. See also fasttime::asPOSIXct for details.
nrows	integer: the maximum number of rows to read in. Negative and other invalid values are ignored.
nsep	index name separator (single character) or NA if no index names are included
strict	logical, if FALSE then dstrsplit will not fail on parsing errors, otherwise input not matching the format (e.g. more columns than expected) will cause an error.
nrowsClasses	integer. Maximum number of rows of data to read to learn column types. Not used when col_types is supplied.
quote	the set of quoting characters as a length 1 vector. To disable quoting altogether, use quote = "". Quoting is only considered for columns read as character.
...	additional parameters to pass to read.csv.raw

Details

See dstrsplit for the details of nsep, sep, and strict.

Value

A data frame containing a representation of the data in the file.

Author(s)

Taylor Arnold and Simon Urbanek

---

readAsRaw *Read binary data in as raw*

Description

readAsRaw takes a connection or file name and reads it into a raw type.

Usage

```
readAsRaw(con, n, nmax, fileEncoding="")
```

Arguments

con	A connection object or a character string naming a file from which to save the output.
n	Expected number of bytes to read. Set to 65536L by default when con is a connection, and set to the file size by default when con is a character string.
nmax	Maximum number of bytes to read; missing of Inf to read in the entire connection.
fileEncoding	When con is a connection, the file encoding to use to open the connection.

Value

readAsRaw returns a raw type which can then be consumed by functions like mstrsplit and dstrsplit.

Author(s)

Taylor Arnold

Examples

```
mm <- model.matrix(~., iris)
f <- file("iris_mm.io", "wb")
writeBin(as.output(mm), f)
close(f)
m <- mstrsplit(readAsRaw("iris_mm.io"), type="numeric", nsep="\t")
head(mm)
head(m)
unlink("iris_mm.io")
```

`which.min.key` *Determine the next key in bytewise order*

Description

`which.min.key` takes either a character vector or a list of strings and returns the location of the element that is lexicographically (using bytewise comparison) the first. In a sense it is `which.min` for strings. In addition, it supports prefix comparisons using a key delimiter (see below).

Usage

```
which.min.key(keys, sep = "|")
```

Arguments

`keys`	character vector or a list of strings to use as input
`sep`	string specifying the key delimiter. Only the first character is used. Can be `""` if the entire string is to be treated as a key.

Details

`which.min.key` considers the prefix of each element in `keys` up to the delimiter specified by `sep`. It returns the index of the element which is lexicographically first among all the elements, using bytewise comparison (i.e. the locale is not used and multi-byte characters are not considered as one character).

If `keys` is a character vector then `NA` elements are treated as non-existent and will never be picked.

If `keys` is a list then only string elements of length > 0 are eligible and `NA`s are not treated specially (hence they will be sorted in just like the `"NA"` string).

Value

scalar integer denoting the index of the lexicographically first element. In case of a tie the lowest index is returned. If there are no eligible elements in `keys` then a zero-length integer vector is returned.

Author(s)

Simon Urbanek

See Also

which.min

Examples

```
which.min.key(c("g","a","b",NA,"z","a"))
which.min.key(c("g","a|z","b",NA,"z|0","a"))
which.min.key(c("g","a|z","b",NA,"z|0","a"), "")
which.min.key(list("X",1,NULL,"F","Z"))
which.min.key(as.character(c(NA, NA)))
which.min.key(NA_character_)
which.min.key(list())
```

---

write.csv.raw *Fast data output to disk*

Description

A fast replacement of `write.csv` and `write.table` which saves the data as a raw vector rather than a character one.

Usage

```
write.csv.raw(x, file = "", append = FALSE, sep = ",", nsep="\t",
              col.names = !is.null(colnames(x)), fileEncoding = "")

write.table.raw(x, file = "", sep = " ", ...)
```

Arguments

`x`	object which is to be saved.
`file`	A connection object or a character string naming a file from which to save the output.
`append`	logical. Only used when file is a character string.
`sep`	field (column) separator.
`nsep`	index name separator (single character) or `NA` if no index names are included
`col.names`	logical. Should a raw of column names be writen.
`fileEncoding`	character string: if non-empty declares the encoding to be used on a file.
`...`	additional parameters to pass to `write.table.raw`.

Details

See `as.output` for the details of how various data types are converted to raw vectors (or character vectors when raw is not available).

Author(s)

Taylor Arnold and Simon Urbanek

Index