Package ‘rock’

January 21, 2024

Title Reproducible Open Coding Kit
Version 0.8.1
Date 2024-01-21
Maintainer Gjalt-Jorn Peters <rock@opens.science>
Description The Reproducible Open Coding Kit ('ROCK', and this package, 'rock')
was developed to facilitate reproducible and open coding, specifically
g geared towards qualitative research methods. Although it is a
general-purpose toolkit, three specific applications have been
implemented, specifically an interface to the 'rENA' package that
implements Epistemic Network Analysis ('ENA'), means to process notes
from Cognitive Interviews ('CIs'), and means to work with decentralized
construct taxonomies ('DCTs'). The 'ROCK' and this 'rock' package are described
in the ROCK book <https://rockbook.org> and more information, such as tutorials,
is available at <https://rock.science>.

BugReports https://gitlab.com/r-packages/rock/-/issues

URL https://rock.opens.science
License GPL-3
Encoding UTF-8
LazyData true
RoxygenNote 7.2.3
Depends R (>= 3.0.0)
Imports data.tree (>= 1.1.0), dplyr (>= 0.7.8), DiagrammeR (>= 1.0.0),
DiagrammeRsvg (>= 0.1), ggplot2 (>= 3.2.0), glue (>= 1.3.0),
graphics (>= 3.0.0), htmltools (>= 0.5.0), markdown (>= 1.1),
purr (>= 0.2.5), stats (>= 3.0.0), utils (>= 3.5.0), yaml (>= 2.2.0), yum (>= 0.1.0)
Suggests covr, googlesheets4, haven (>= 2.4), justifier (>= 0.2),
knitr, limonaid, openxlsx (>= 4.2), pdftools, preregr (>= 0.1.9), readxl, rmarkdown, rvest, rstudioapi, striprtf,
testthat, writexl, XLConnect, xml2

VignetteBuilder knitr
NeedsCompilation: no

Author: Gjalt-Jorn Peters [aut, cre] (<https://orcid.org/0000-0002-0336-9589>), Szilvia Zorgo [aut, ctb] (<https://orcid.org/0000-0002-6916-2097>)

Repository: CRAN

Date/Publication: 2024-01-21 15:40:02 UTC

**R topics documented:**

- add_html_tags .................................................. 4
- apply_graph_theme ............................................. 5
- base30toNumeric ............................................... 6
- cat0 .............................................................. 7
- checkPkgs .......................................................... 7
- ci_get_item ...................................................... 8
- ci_heatmap ......................................................... 9
- ci_import_nrm_spec ............................................. 10
- cleaned_source_to Utterance_vector .......................... 11
- clean_source .................................................... 12
- codebook_fromSpreadsheet ................................... 17
- codebook_to_pdf ............................................... 18
- codeIds_to_codePaths ......................................... 19
- codePaths_to_namedVector .................................... 20
- code_freq_hist .................................................. 21
- code_source ....................................................... 22
- codingSchemes_get_all ........................................ 24
- collapse_occurrences ......................................... 25
- collect_coded_fragments ...................................... 26
- compress_with_sum ............................................. 29
- convertToNumeric ............................................... 30
- convert_df_to_source .......................................... 30
- create_codingScheme .......................................... 35
- create_cooccurrence_matrix .................................. 36
- css ............................................................... 37
- doc_to_txt ........................................................ 38
- exampleCodebook_1 ............................................. 39
- expand_attributes .............................................. 39
- exportToHTML .................................................. 41
- export_codes_to_txt ........................................... 42
- export_mergedSourceDF_to_csv ............................... 43
- export_to_html ................................................ 45
- extract_codings_by_coderId .................................. 46
- form_to_rmd_template ......................................... 47
- generate_uids ................................................... 49
- generic_recoding ............................................... 50
- get_childCodeIds .............................................. 51
- get_dataframe_from_nested_list .............................. 52
- get_source_filter ............................................ 53
R topics documented:

get_state_transition_df ........................................... 54
get_state_transition_dot ........................................... 55
get_state_transition_table ......................................... 56
get_utterances_and_codes_from_source .......................... 57
get_vectors_from_nested_list ...................................... 58
heading .................................................................. 59
heatmap_basic ............................................................. 59
inspect_coded_sources ............................................... 60
load_source ................................................................ 61
mask_source ............................................................... 63
match_consecutive_delimiters ........................................ 65
merge_sources ............................................................. 66
number_as_xl_date ....................................................... 68
opts ..................................................................... 68
parsed_sources_to_ena_network ..................................... 70
parse_source ............................................................. 72
parse_source_by_coderId .............................................. 75
prepend_ids_to_source .................................................. 76
prereg_initialize ......................................................... 78
print.rock_graphList ................................................... 79
qna_to_tlm .................................................................. 79
rbind_dfs .................................................................. 80
rbind_df_list ............................................................... 81
read_spreadsheet ....................................................... 81
recode_addChildCodes ............................................... 82
recode_delete ............................................................. 84
recode_merge .............................................................. 86
recode_move ............................................................... 88
recode_rename ............................................................ 90
recode_split ............................................................... 91
repeatStr ................................................................. 94
resultsOverview_allCodedFragments .............................. 94
rock .................................................................... 96
root_from_codePaths ................................................... 97
rpe_create_source_with_items ...................................... 97
save_workspace .......................................................... 99
show_attribute_table .................................................. 100
show_fullyMergedCodeTrees ....................................... 101
show_inductive_code_tree .......................................... 102
split_long_lines ......................................................... 103
stripCodePathRoot ..................................................... 104
syncing_df_compress .................................................. 104
syncing_df_expand ..................................................... 106
sync_streams ............................................................. 107
sync_vector ................................................................ 109
template_ci_heatmap_1_to_pdf ..................................... 110
template_codebook_to_pdf .......................................... 111
vecTxt .................................................................... 112
Description

This function adds HTML tags to a source to allow pretty printing/viewing.

Usage

```r
add_html_tags(
  x,
  context = NULL,
  codeClass = rock::opts$get(codeClass),
  codeValueClass = rock::opts$get(codeValueClass),
  idClass = rock::opts$get(idClass),
  sectionClass = rock::opts$get(sectionClass),
  uidClass = rock::opts$get(uidClass),
  contextClass = rock::opts$get(contextClass),
  utteranceClass = rock::opts$get(utteranceClass)
)
```

Arguments

- `x` A character vector with the source
- `context` Optionally, lines to pass the contextClass
- `codeClass`, `codeValueClass`, `idClass`, `sectionClass`, `uidClass`, `contextClass`, `utteranceClass`
  The classes to use for, respectively, codes, code values, class instance identifiers (such as case identifiers or coder identifiers), section breaks, utterance identifiers, context, and full utterances. All `<span>` elements except for the full utterances, which are placed in `<div>` elements.

Value

The character vector with the replacements made.

Examples

```r
### Add tags to a mini example source
add_html_tags("[[cid=participant1]]
This is something this participant may have said.
Just like this. [[thisIsACode]]
---paragraph-break---
And another utterance.");
```
apply_graph_theme

---

apply_graph_theme

Apply multiple DiagrammeR global graph attributes

Description

Apply multiple DiagrammeR global graph attributes

Usage

apply_graph_theme(graph, ...)

Arguments

graph The DiagrammeR::DiagrammeR graph to apply the attributes to.

... One or more character vectors of length three, where the first element is the attribute, the second the value, and the third, the attribute type (graph, node, or edge).

Value

The DiagrammeR::DiagrammeR graph.

Examples

exampleSource <- '---
codes:
  -
    id: parentCode
    label: Parent code
    children:
      -
        id: childCode1
      -
        id: childCode2
      -
        id: childCode3
    label: Child Code
    parentId: parentCode
    children: [grandChild1, grandChild2]
---
';
parsedSource <-
  parse_source(text=exampleSource);
miniGraph <-
  apply_graph_theme(data.tree::ToDiagrammeRGraph(parsedSource$deductiveCodeTrees),
                    c("color", "#0000AA", "node"),
                    c("shape", "triangle", "node"),
                    c("fontcolor", "#FF0000", "node"));
### This line should be run when executing this example as test, because
### rendering a DiagrammeR graph takes quite long
## Not run:
DiagrammeR::render_graph(miniGraph);
## End(Not run)

### base30toNumeric

#### Conversion between base10 and base30

**Description**

The conversion functions from base10 to base30 and vice versa are used by the `generate_uids()` functions.

**Usage**

```r
base30toNumeric(x)
numericToBase30(x)
```

**Arguments**

- `x` The vector to convert (numeric for `numericToBase30`, character for `base30toNumeric`).

**Details**

The symbols to represent the 'base 30' system are the 0-9 followed by the alphabet without vowels but including the y. This vector is available as `base30`.

**Value**

The converted vector (numeric for `base30toNumeric`, character for `numericToBase30`).

**Examples**

```r
numericToBase30(654321);
base30toNumeric(numericToBase30(654321));
```
cat0

Concatenate to screen without spaces

Description

The cat0 function is to cat what paste0 is to paste; it simply makes concatenating many strings without a separator easier.

Usage

\[
\text{cat0}(\ldots, \text{sep} = \"\"igr)
\]

Arguments

\(\ldots\) The character vector(s) to print; passed to cat.

\(\text{sep}\) The separator to pass to cat, of course, \"\" by default.

Value

Nothing (invisible NULL, like cat).

Examples

\[
\text{cat0}\("\text{The first variable is}\), \text{names(mtcars)}[1], \".\text{\}"\bigr);
\]

cat0

checkPkg

Check for presence of a package

Description

This function efficiently checks for the presence of a package without loading it (unlike library() or require()). This is useful to force yourself to use the package::function syntax for addressing functions; you can make sure required packages are installed, but their namespace won’t attach to the search path.

Usage

\[
\text{checkPkg}\(
\begin{array}{l}
\ldots, \\
\text{install} = \text{FALSE}, \\
\text{load} = \text{FALSE}, \\
\text{repos} = \"https://cran.rstudio.com\" \\
\end{array}
\)
\]
ci_get_item

Arguments

... A series of packages. If the packages are named, the names are the package names, and the values are the minimum required package versions (see the second example).

install Whether to install missing packages from repos.

load Whether to load packages (which is exactly not the point of this function, but hey, YMMV).

repos The repository to use if installing packages; default is the RStudio repository.

Value

Invisibly, a vector of the available packages.

Examples

rock::checkPkgs('base');

### Require a version
rock::checkPkgs(rock = "0.5.0");

### This will show the error message
tryCatch(
  rock::checkPkgs(
    base = "99",
    stats = "42.5",
    ufs = 20
  ),
  error = print
);

ci_get_item Get an item in a specific language

Description

This function takes a Narrative Response Model specification as used in NRM-based cognitive interviews, and composes an item based on the specified template for that item, the specified stimuli, and the requested language.

Usage

  ci_get_item(nrm_spec, item_id, language)
ci_heatmap

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nrm_spec</td>
<td>The Narrative Response Model specification.</td>
</tr>
<tr>
<td>item_id</td>
<td>The identifier of the requested item.</td>
</tr>
<tr>
<td>language</td>
<td>The language of the stimuli.</td>
</tr>
</tbody>
</table>

Value

A character value with the item.

---

**ci_heatmap**

Create a heatmap showing issues with items

Description

When conducting cognitive interviews, it can be useful to quickly inspect the code distributions for each item. These heatmaps facilitate that process.

Usage

```r
ci_heatmap(
  x,
  nrmSpec = NULL,
  language = nrmSpec$defaultLanguage,
  wrapLabels = 80,
  itemOrder = NULL,
  itemLabels = NULL,
  itemIdentifier = "uiid",
  codingScheme = "peterson",
  itemlab = NULL,
  codelab = NULL,
  freqlab = "Count",
  plotTitle = "Cognitive Interview Heatmap",
  fillScale = ggplot2::scale_fill_viridis_c(),
  theme = ggplot2::theme_minimal()
)
```

Arguments

- **x**: The object with the parsed coded source(s) as resulting from a call to `parse_source()` or `parse_sources()`.
- **nrmSpec**: Optionally, an imported Narrative Response Model specification, as imported with `ci_import_nrm_spec()`, which will then be used to obtain the item labels.
- **language**: If `nrmSpec` is specified, the language to use.
- **wrapLabels**: Whether to wrap the labels; if not NULL, the number of character to wrap at.
Instead of specifying an NRM specification, you can also directly specify the item order and item labels. `itemOrder` is a character vector of item identifiers, and `itemLabels` is a named character vector of item labels, where each value's name is the corresponding item identifier. If `itemLabels` is provided but `itemOrder` is not, the order of the `itemLabel` is used.

**itemIdentifier**  
The column identifying the items; the class instance identifier prefix, e.g. if item identifiers are specified as `[[uiid:familySize_7djdy62d]]`, the `itemIdentifier` to pass here is "uiid".

**codingScheme**  
The coding scheme, either as a string if it represents one of the cognitive interview coding schemes provided with the `rock` package, or as a coding scheme resulting from a call to `create_codingScheme()`.

**itemlab, codelab, freqlab**  
Labels to use for the item and code axes and for the frequency color legend (NULL to omit the label).

**plotTitle**  
The title to use for the plot.

**fillScale**  
Convenient way to specify the fill scale (the colours)

**theme**  
Convenient way to specify the `ggplot2::ggplot()` theme.

**Value**  
The heatmap as a `ggplot2` plot.

**Examples**

```r
examplePath <- file.path(system.file(package="rock"), 'extdata');
parsedCI <- rock::parse_source(  
  file.path(examplePath,  
    "ci_example_1.rock")  
);

rock::ci_heatmap(parsedCI,  
  codingScheme = "peterson");
```

---

**ci_import_nrm_spec**  
**Import a Narrative Response Model specification**

**Description**

Narrative Response Models are a description of the theory of how a measurement instrument that measures a psychological construct works, geared towards conducting cognitive interviews to verify the validity of that measurement instrument. One a Narrative Response Model has been imported, it can be used to generate interview schemes, overview of each item's narrative response model, and combined with coded cognitive interview notes or transcripts.
Usage

```r
ci_import_nrm_spec(
  x,
  read_ss_args = list(exportGoogleSheet = TRUE),
  defaultLanguage = NULL,
  silent = rock::opts$get("silent")
)
```

```r
## S3 method for class 'rock_ci_nrm'
print(x, ...)
```

Arguments

- **x**: A path to a file or an URL to a Google Sheet, passed to `read_spreadsheet()`.
- **read_ss_args**: A named list with arguments to pass to `read_spreadsheet()`.
- **defaultLanguage**: Language to set as default language (by default, i.e. if NULL, the first language is used).
- **silent**: Whether to be silent or chatty.
- **...**: Additional arguments are ignored.

Value

A `rock_ci_nrm` object.

---

**cleaned_source_to_utterance_vector**

*Convert a character vector into an utterance vector*

Description

Utterance vectors are split by the utterance marker. Note that if `x` has more than one element, the separate elements will remain separate.

Usage

```r
cleaned_source_to_utterance_vector(
  x,
  utteranceMarker = rock::opts$get("utteranceMarker"),
  fixed = FALSE,
  perl = TRUE
)
```
Arguments

- `x` The character vector.
- `utteranceMarker` The utterance marker (by default, a newline character conform the ROCK standard).
- `fixed` Whether the `utteranceMarker` is a regular expression.
- `perl` If the `utteranceMarker` is a regular expression, whether it is a perl regular expression.

Value

A character vector with separate utterances, split by `utteranceMarker`.

Examples

```R
cleaned_source_to_utterance_vector("first
second
third");
```

Description

These functions can be used to 'clean' one or more sources or perform search and replace taks. Cleaning consists of two operations: splitting the source at utterance markers, and conducting search and replaces using regular expressions.

Usage

```R
clean_source(
  input,
  output = NULL,
  replacementsPre = rock::opts$get("replacementsPre"),
  replacementsPost = rock::opts$get("replacementsPost"),
  extraReplacementsPre = NULL,
  extraReplacementsPost = NULL,
  removeNewlines = FALSE,
  removeTrailingNewlines = TRUE,
  rlWarn = rock::opts$get(rlWarn),
  utteranceSplits = rock::opts$get("utteranceSplits"),
  preventOverwriting = rock::opts$get("preventOverwriting"),
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)
```
clean_sources(
  input,
  output,
  outputPrefix = "",
  outputSuffix = "_cleaned",
  recursive = TRUE,
  filenameRegex = ".*",
  replacementsPre = rock::opts$get(replacementsPre),
  replacementsPost = rock::opts$get(replacementsPost),
  extraReplacementsPre = NULL,
  extraReplacementsPost = NULL,
  removeNewlines = FALSE,
  utteranceSplits = rock::opts$get(utteranceSplits),
  preventOverwriting = rock::opts$get(preventOverwriting),
  encoding = rock::opts$get(encoding),
  silent = rock::opts$get(silent)
)

search_and_replace_in_source(
  input,
  replacements = NULL,
  output = NULL,
  preventOverwriting = TRUE,
  encoding = "UTF-8",
  rlWarn = rock::opts$get(rlWarn),
  silent = FALSE
)

search_and_replace_in_sources(
  input,
  output,
  replacements = NULL,
  outputPrefix = "",
  outputSuffix = "_postReplacing",
  preventOverwriting = rock::opts$get("preventOverwriting"),
  recursive = TRUE,
  filenameRegex = ".*",
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)

wordwrap_source(
  input,
  output = NULL,
  length = 60,
  removeNewlines = FALSE,
  removeTrailingNewlines = TRUE,
rlWarn = rock::opts$get(rlWarn),
preventOverwriting = rock::opts$get("preventOverwriting"),
encoding = rock::opts$get(encoding),
silent = rock::opts$get(silent),
utteranceMarker = rock::opts$get("utteranceMarker")
)

Arguments

input For clean_source and search_and_replace_in_source, either a character vector containing the text of the relevant source or a path to a file that contains the source text; for clean_sources and search_and_replace_in_sources, a path to a directory that contains the sources to clean.

output For clean_source and search_and_replace_in_source, if not NULL, this is the name (and path) of the file in which to save the processed source (if it is NULL, the result will be returned visibly). For clean_sources and search_and_replace_in_sources, output is mandatory and is the path to the directory where to store the processed sources. This path will be created with a warning if it does not exist. An exception is if "same" is specified - in that case, every file will be written to the same directory it was read from.

replacementsPre, replacementsPost Each is a list of two-element vectors, where the first element in each vector contains a regular expression to search for in the source(s), and the second element contains the replacement (these are passed as perl regular expressions; see regex for more information). Instead of regular expressions, simple words or phrases can also be entered of course (since those are valid regular expressions). replacementsPre are executed before the utteranceSplits are applied; replacementsPost afterwards.

extraReplacementsPre, extraReplacementsPost To perform more replacements than the default set, these can be conveniently specified in extraReplacementsPre and extraReplacementsPost. This prevents you from having to manually copypaste the list of defaults to retain it.

removeNewlines Whether to remove all newline characters from the source before starting to clean them. Be careful: if the source contains YAML fragments, these will also be affected by this, and will probably become invalid!

removeTrailingNewlines Whether to remove trailing newline characters (i.e. at the end of a character value in a character vector);

rlWarn Whether to let readLines() warn, e.g. if files do not end with a newline character.

utteranceSplits This is a vector of regular expressions that specify where to insert breaks between utterances in the source(s). Such breaks are specified using utteranceMarker.

preventOverwriting Whether to prevent overwriting of output files.

encoding The encoding of the source(s).

silent Whether to suppress the warning about not editing the cleaned source.
outputPrefix, outputSuffix
The prefix and suffix to add to the filenames when writing the processed files to disk.

recursive
Whether to search all subdirectories (TRUE) as well or not.

filenameRegex
A regular expression to match against located files; only files matching this regular expression are processed.

replacements
The strings to search & replace, as a list of two-element vectors, where the first element in each vector contains a regular expression to search for in the source(s), and the second element contains the replacement (these are passed as perl regular expressions; see regex for more information). Instead of regular expressions, simple words or phrases can also be entered of course (since those are valid regular expressions).

length
At how many characters to word wrap.

utteranceMarker
The character(s) between utterances (i.e. marking where one utterance ends and the next one starts). By default, this is a line break, and only change this if you know what you are doing.

Details
The cleaning functions, when called with their default arguments, will do the following:

- Double periods (..) will be replaced with single periods (.)
- Four or more periods (.... or ....) will be replaced with three periods
- Three or more newline characters will be replaced by one newline character (which will become more, if the sentence before that character marks the end of an utterance)
- All sentences will become separate utterances (in a semi-smart manner; specifically, breaks in speaking, if represented by three periods, are not considered sentence ends, whereas ellipses ("..." or unicode 2026, see the example) are.
- If there are comma’s without a space following them, a space will be inserted.

The cleaning functions, when called with their default arguments, will do the following:

- Double periods (..) will be replaced with single periods (.)
- Four or more periods (.... or ....) will be replaced with three periods
- Three or more newline characters will be replaced by one newline character (which will become more, if the sentence before that character marks the end of an utterance)
- All sentences will become separate utterances (in a semi-smart manner; specifically, breaks in speaking, if represented by three periods, are not considered sentence ends, whereas ellipses ("..." or unicode 2026, see the example) are.
- If there are comma’s without a space following them, a space will be inserted.

Value
A character vector for clean_source, or a list of character vectors, for clean_sources.
A character vector for clean_source, or a list of character vectors, for clean_sources.
Examples

```r
eexampleSource <-
  "Do you like icecream?

Well, that depends\u2026 Sometimes, when it's..... Nice. Then I do, but otherwise... not really, actually."

### Default settings:
cat(clean_source(exampleSource));

### First remove existing newlines:
cat(clean_source(exampleSource, removeNewlines=TRUE));

### Example with a YAML fragment
eexampleWithYAML <-
c(  
  "Do you like icecream?",
  ""," ",
  "Well, that depends\u2026 Sometimes, when it's..... Nice.",
  "Then I do.",
  "but otherwise... not really, actually.",
  "",
  "---",
  "This acts as some YAML. So this won't be split.",
  "Not real YAML, mind... It just has the delimiters, really.",
  "---",
  "This is an utterance again."
);

cat(
  rock::clean_source(
    exampleWithYAML  
  ),
  sep="\n"
);

eexampleSource <-
  "Do you like icecream?

Well, that depends\u2026 Sometimes, when it's..... Nice. Then I do, but otherwise... not really, actually."

### Simple text replacements:
cat(search_and_replace_in_source(exampleSource,
  replacements=list(c("\u2026", "..."),
                    c(Nice", "Great"))));

### Using a regular expression to capitalize all words following
### a period:
cat(search_and_replace_in_source(exampleSource,
   replacements=list(c("\.(\s*)([a-z])", "\s\1\u2026\2"))));

exampleSource <-
"Do you like icecream?

Well, that depends\u2026 Sometimes, when it's..... Nice. Then I do,
but otherwise... not really, actually."

### Default settings:
cat(clean_source(exampleSource));

### First remove existing newlines:
cat(clean_source(exampleSource,
   removeNewlines=TRUE));

### Example with a YAML fragment
exampleWithYAML <-
c(  "Do you like icecream?",
   "",
   "",
   "Well, that depends\u2026 Sometimes, when it's..... Nice.",
   "Then I do,",
   "but otherwise... not really, actually.",
   "",
   "---",
   "This acts as some YAML. So this won't be split.",
   "Not real YAML, mind... It just has the delimiters, really.",
   "---",
   "This is an utterance again."
);

cat(
   rock::clean_source(
      exampleWithYAML
   ),
   sep="\n"
);

---

Import a code book specification from a spreadsheet

**Description**

Import a code book specification from a spreadsheet
Usage

codebook_fromSpreadsheet(
    x,
    localBackup = NULL,
    exportGoogleSheet = TRUE,
    xlsxPkg = c("rw_xl", "openxlsx", "XLConnect"),
    silent = rock::opts$.<get("silent")
)

Arguments

x The URL or path to a file.
localBackup If not NULL, a valid filename to write a local backup to.
exportGoogleSheet If x is a URL to a Google Sheet, instead of using the googlesheets4 package to
download the data, by passing exportGoogleSheet=TRUE, an export link will
be produced and the data will be downloaded as Excel spreadsheet.
xlsxPkg Which package to use to work with Excel spreadsheets.
silent Whether to be silent or chatty.

Value

The code book specification as a rock code book object

Examples

### This requires an active internet connection
if (FALSE) {
    gs_url <- paste0(
        "https://docs.google.com/spreadsheets/d/",
        "1gVx5uhYzcTH6Jq7AYmsLVH5BaYAT-23c72Z3F4jmps"
    );
    codebook <- rock::codebook_fromSpreadsheet(gs_url);
}

---

**codebook_to_pdf**  
*Convert a codebook specification to PDF*

Description

Use this function to export your codebook specification to a PDF file. To embed it in an R Mark-
down file, use !!! CREATE rock::knit_codebook() !!!
codeIds_to_codePaths

Usage

codebook_to_pdf(
  x,
  file,
  author = NULL,
  headingLevel = 1,
  silent = rock::opts$get("silent")
)

Arguments

  x                      The codebook object (as produced by a call to codebook_fromSpreadsheet()).
  file                   The filename to save the codebook to.
  author                 The author to specify in the PDF.
  headingLevel           The level of the top-most headings.
  silent                 Whether to be silent or chatty.

Value

  x, invisibly

Examples

  ### Use a temporary file to write to
tmpFile <- tempfile(fileext = ".pdf");

  ### Load an example (pre)registration specification
data("exampleCodebook_1", package = "rock");

  rock::codebook_to_pdf(
    exampleCodebook_1,
    file = tmpFile
  );

---

codeIds_to_codePaths  Replace code identifiers with their full paths

Description

This function replaces the column names in the mergedSourceDf data frame in a rock_parsedSource or rock_parsedSources object with the full paths to those code identifiers.
codeIds_to_codePaths

Usage

```r
codeIds_to_codePaths(
  x,
  stripRootsFromCodePaths = rock::opts$get("stripRootsFromCodePaths")
)
```

Arguments

- `x`: A `rock_parsedSource` or `rock_parsedSources` object as returned by a call to `parse_source()` or `parse_sources()`.
- `stripRootsFromCodePaths`: Whether to strip the roots first (i.e. the type of code).

Value

An adapted `rock_parsedSource` or `rock_parsedSources` object.

codePaths_to_namedVector

Get a vector to find the full paths based on the leaf code identifier

Description

This function names a vector with the leaf code using the `codeTreeMarker` stored in the `opts` object as marker.

Usage

```
codePaths_to_namedVector(x)
```

Arguments

- `x`: A vector of code paths.

Value

The named vector of code paths.

Examples

```r
codePaths_to_namedVector(
  c("codes>reason>parent_feels",
     "codes>reason>child_feels")
);
```
code_freq_hist

Create a frequency histogram for codes

Description

Create a frequency histogram for codes

Usage

code_freq_hist(
  x,
  codes = ".*",
  sortByFreq = "decreasing",
  forceRootStripping = FALSE,
  trimSourceIdentifiers = 20,
  ggplot2Theme = ggplot2::theme(legend.position = "bottom"),
  silent = rock::opts$get("silent")
)

Arguments

x A parsed source(s) object.
codes A regular expression to select codes to include.
sortByFreq Whether to sort by frequency decreasingly (decreasing, the default), increasingly (increasing), or alphabetically (NULL).
forceRootStripping Force the stripping of roots, even if they are different.
trimSourceIdentifiers If not NULL, the number of character to trim the source identifiers to.
ggplot2Theme Can be used to specify theme elements for the plot.
silent Whether to be chatty or silent.

Value

a ggplot2::ggplot().

Examples

### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <-
  file.path(examplePath, "example-1.rock");
### Load example source
loadedExample <- rock::parse_source(exampleFile);

### Show code frequencies
code_freq_hist(loadedExample);

---

code_source  

Add one or more codes to one or more sources

**Description**

These functions add codes to one or more sources that were read with one of the `loading_sources` functions.

**Usage**

code_source(
  input,
  codes,
  indices = NULL,
  output = NULL,
  decisionLabel = NULL,
  justification = NULL,
  justificationFile = NULL,
  preventOverwriting = rock::opts$get("preventOverwriting"),
  rlWarn = rock::opts$get(rlWarn),
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)

code_sources(
  input,
  codes,
  output = NULL,
  indices = NULL,
  outputPrefix = "",
  outputSuffix = "_coded",
  decisionLabel = NULL,
  justification = NULL,
  justificationFile = NULL,
  recursive = TRUE,
  filenameRegex = ".*",
  preventOverwriting = rock::opts$get("preventOverwriting"),
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)
Arguments

input  The source, or list of sources, as produced by one of the loading_sources functions.
codes  A named character vector, where each element is the code to be added to the matching utterance, and the corresponding name is either an utterance identifier (in which case the utterance with that identifier will be coded with that code), a code (in which case all utterances with that code will be coded with the new code as well), a digit (in which case the utterance at that line number in the source will be coded with that code), or a regular expression, in which case all utterances matching that regular expression will be coded with that source. If specifying an utterance ID or code, make sure that the code delimiters are included (normally, two square brackets).
indices If input is a source as loaded by loading_sources, indices can be used to pass a logical vector of the same length as input that indicates to which utterance the code in codes should be applied. Note that if indices is provided, only the first element of codes is used, and its name is ignored.
output  If specified, the coded source will be written here.
decisionLabel  A description of the (coding) decision that was taken.
justification  The justification for this action.
justificationFile  If specified, the justification is appended to this file. If not, it is saved to the justifier::workspace(). This can then be saved or displayed at the end of the R Markdown file or R script using justifier::save_workspace().
preventOverwriting  Whether to prevent overwriting existing files.
rlWarn  Whether to let readLines() warn, e.g. if files do not end with a newline character.
encoding  The encoding to use.
silent  Whether to be chatty or quiet.
outputPrefix, outputSuffix  A prefix and/or suffix to prepend and/or append to the filenames to distinguish them from the input filenames.
recursive  Whether to also read files from all subdirectories of the input directory
filenameRegex  Only input files matching this patterns will be read.

Value

Invisibly, the coded source object.

Examples

```r
### Get path to example source
examplePath <-
system.file("extdata", package="rock");
```
codingSchemes_get_all

Convenience function to get a list of all available coding schemes

Description
Convenience function to get a list of all available coding schemes

Usage

codingSchemes_get_all()
collapse_occurrences

Value
A list of all available coding schemes

Examples
rock::codingSchemes_get_all();

collapse_occurrences  Collapse the occurrences in utterances into groups

Description
This function collapses all occurrences into groups sharing the same identifier, by default the stanzaId identifier ([[sid=..]]).

Usage
collapse_occurrences(
  parsedSource,
  collapseBy = "stanzaId",
  columns = NULL,
  logical = FALSE
)

Arguments
parsedSource  The parsed sources as provided by parse_source()
collapseBy  The column in the sourceDf (in the parsedSource object) to collapse by (i.e. the column specifying the groups to collapse).
columns  The columns to collapse; if unspecified (i.e. NULL), all codes stored in the code object in the codings object in the parsedSource object are taken (i.e. all used codes in the parsedSource object).
logical  Whether to return the counts of the occurrences (FALSE) or simply whether any code occurred in the group at all (TRUE).

Value
A dataframe with one row for each value of of collapseBy and columns for collapseBy and each of the columns, with in the cells the counts (if logical is FALSE) or TRUE or FALSE (if logical is TRUE).
Examples

```r
### Get path to example source
eexampleFile <-
  system.file("extdata", "example-1.rock", package="rock");

### Parse example source
parsedExample <-
  rock::parse_source(exampleFile);

### Collapse logically, using a code (either occurring or not):
collapsedExample <-
  rock::collapse_occurrences(parsedExample,
    collapseBy = "childCode1");

### Show result: only two rows left after collapsing,
### because 'childCode1' is either 0 or 1:
collapsedExample;

### Collapse using weights (i.e. count codes in each segment):
collapsedExample <-
  rock::collapse_occurrences(parsedExample,
    collapseBy = "childCode1",
    logical=FALSE);
```

---

**collect_coded_fragments**

Create an overview of coded fragments

---

**Description**

Collect all coded utterances and optionally add some context (utterances before and utterances after) to create an overview of all coded fragments per code.

**Usage**

```r
collect_coded_fragments(
  x,
  codes = ".*",
  context = 0,
  includeDescendents = FALSE,
  attributes = NULL,
  heading = NULL,
  headingLevel = 3,
  add_html_tags = TRUE,
  cleanUtterances = FALSE,
  output = NULL,
```
collect_coded_fragments

outputViewer = "viewer",
template = "default",
rawResult = FALSE,
includeCSS = TRUE,

codeHeadingFormatting = rock::opts$get("codeHeadingFormatting"),
includeBootstrap = rock::opts$get("includeBootstrap"),
preventOverwriting = rock::opts$get("preventOverwriting"),
silent = rock::opts$get("silent")
)

Arguments

x The parsed source(s) as provided by rock::parse_source or rock::parse_sources.
codes The regular expression that matches the codes to include, or a character vector with codes or regular expressions for codes (which will be prepended with "^" and appended with "$", and then concatenated using "|") as a separator, to create a regular expression matching all codes).
context How many utterances before and after the target utterances to include in the fragments. If two values, the first is the number of utterances before, and the second, the number of utterances after the target utterances.
includeDescendents Whether to also collect the fragments coded with descendent codes (i.e. child codes, ‘grand child codes’, etc; in other words, whether to collect the fragments recursively).
attributes To only select coded utterances matching one or more values for one or more attributes, pass a list where every element’s name is a valid (i.e. occurring) attribute name, and every element is a character value with a regular expression specifying all values for that attribute to select.
heading Optionally, a title to include in the output. The title will be prefixed with headingLevel hashes (#), and the codes with headingLevel+1 hashes. If NULL (the default), a heading will be generated that includes the collected codes if those are five or less. If a character value is specified, that will be used. To omit a heading, set to anything that is not NULL or a character vector (e.g. FALSE). If no heading is used, the code prefix will be headingLevel hashes, instead of headingLevel+1 hashes.
headingLevel The number of hashes to insert before the headings.
add_html_tags Whether to add HTML tags to the result.
cleanUtterances Whether to use the clean or the raw utterances when constructing the fragments (the raw versions contain all codes). Note that this should be set to FALSE to have add_html_tags be of the most use.
output Here, a path and filename can be provided where the result will be written. If provided, the result will be returned invisibly.
outputViewer If showing output, where to show the output: in the console (outputViewer = 'console') or in the viewer (outputViewer = 'viewer'), e.g. the RStudio viewer. You’ll usually want the latter when outputting HTML, and otherwise the former. Set to FALSE to not output anything to the console or the viewer.
collect_coded_fragments

```
template
rawResult
includeCSS
codeHeadingFormatting
includeBootstrap
preventOverwriting
silent
```

**Details**

By default, the output is optimized for inclusion in an R Markdown document. To optimize output for the R console or a plain text file, without any HTML codes, set `add_html_tags` to FALSE, and potentially set `cleanUtterances` to only return the utterances, without the codes.

**Value**

Either a list of character vectors, or a single character value.

**Examples**

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <-
  file.path(
    examplePath, "example-1.rock"
  );

### Parse single example source
parsedExample <-
  rock::parse_source(
    exampleFile
  );

### Show organised coded fragments in Markdown
cat(
  rock::collect_coded_fragments(
    parsedExample
  )
);
```
### Only for the codes containing 'Code2'

cat(
  rock::collect_coded_fragments(
    parsedExample,
    'Code2'
  )
);

---

**compress_with_sum**  
*Vector compression helper functions*

**Description**

These functions can help when compressing vectors. They always compress their input \(x\) into a single element by various means.

**Usage**

```r
compress_with_sum(x)
compress_with_or(x)
```

**Arguments**

- **x**  
  The vector to compress

**Details**

`compress_with_sum` computes the sum of the elements, doing its best to convert all input values to numeric values. `compress_with_or` returns 0 if all elements are FALSE, 0, NA or empty character values (""), and 1 otherwise.

**Value**

The compressed element

**Examples**

```r
rock::compress_with_sum(c(1, '1', 0));
rock::compress_with_or(c(1, '1', 0));
rock::compress_with_or(c(0, '', 0, FALSE));
```
**convertToNumeric**  

*Conveniently convert vectors to numeric*

**Description**

Tries to ‘smartly’ convert factor and character vectors to numeric.

**Usage**

```r
convertToNumeric(vector, byFactorLabel = FALSE)
```

**Arguments**

- `vector`: The vector to convert.
- `byFactorLabel`: When converting factors, whether to do this by their label value (TRUE) or their level value (FALSE).

**Value**

The converted vector.

**Examples**

```r
rock::convertToNumeric(as.character(1:8));
```

---

**convert_df_to_source**  

*Convert ‘rectangular’ or spreadsheet-format data to one or more sources*

**Description**

These functions first import data from a ‘data format’, such as spreadsheets in .xlsx format, comma-separated values files (.csv), or SPSS data files (.sav). You can also just use R data frames (imported however you want). These functions then use the columns you specified to convert these data to one (oneFile=TRUE) or more (oneFile=FALSE) rock source file(s), optionally including class instance identifiers (such as case identifiers to identify participants, or location identifiers, or moment identifiers, etc) and using those to link the utterances to attributes from columns you specified. You can also precode the utterances with codes you specify (if you ever would want to for some reason).
Usage

```r
convert_df_to_source(
  data, 
  output = NULL, 
  omit_empty_rows = TRUE, 
  cols_to_utterances = NULL, 
  cols_to_ciids = NULL, 
  cols_to_codes = NULL, 
  cols_to_attributes = NULL, 
  utterance_classId = NULL, 
  oneFile = TRUE, 
  cols_to_sourceFilename = cols_to_ciids, 
  cols_in_sourceFilename_sep = "=", 
  sourceFilename_prefix = "source_", 
  sourceFilename_suffix = "", 
  ciid_labels = NULL, 
  ciid_separator = "=",
  attributesFile = NULL, 
  preventOverwriting = rock::opts$g(\text{preventOverwriting}), 
  encoding = rock::opts$g(\text{encoding}), 
  silent = rock::opts$g(\text{silent})
)
```

```r
convert_csv_to_source(
  file, 
  importArgs = NULL, 
  omit_empty_rows = TRUE, 
  output = NULL, 
  cols_to_utterances = NULL, 
  cols_to_ciids = NULL, 
  cols_to_codes = NULL, 
  cols_to_attributes = NULL, 
  oneFile = TRUE, 
  cols_to_sourceFilename = cols_to_ciids, 
  cols_in_sourceFilename_sep = "=", 
  sourceFilename_prefix = "source_", 
  sourceFilename_suffix = "", 
  ciid_labels = NULL, 
  ciid_separator = "=",
  attributesFile = NULL, 
  preventOverwriting = rock::opts$g(\text{preventOverwriting}), 
  encoding = rock::opts$g(\text{encoding}), 
  silent = rock::opts$g(\text{silent})
)
```

```r
convert_csv2_to_source(
  file, 
  importArgs = NULL, 
  omit_empty_rows = TRUE, 
  output = NULL, 
  cols_to_utterances = NULL, 
  cols_to_ciids = NULL, 
  cols_to_codes = NULL, 
  cols_to_attributes = NULL, 
  oneFile = TRUE, 
  cols_to_sourceFilename = cols_to_ciids, 
  cols_in_sourceFilename_sep = "=", 
  sourceFilename_prefix = "source_", 
  sourceFilename_suffix = "", 
  ciid_labels = NULL, 
  ciid_separator = "=",
  attributesFile = NULL, 
  preventOverwriting = rock::opts$g(\text{preventOverwriting}), 
  encoding = rock::opts$g(\text{encoding}), 
  silent = rock::opts$g(\text{silent})
)
```
convert_df_to_source

omit_empty_rows = TRUE,
output = NULL,
cols_to_utterances = NULL,
cols_to_ciids = NULL,
cols_to_codes = NULL,
cols_to_attributes = NULL,
oneFile = TRUE,
cols_to_sourceFilename = cols_to_ciids,
cols_in_sourceFilename_sep = "=",
sourceFilename_prefix = "source_",
sourceFilename_suffix = "",
ciid_labels = NULL,
Ciid_separator = "=",
attributesFile = NULL,
preventOverwriting = rock::opts$getattr(preventOverwriting),
encoding = rock::opts$getattr(encoding),
silent = rock::opts$getattr(silent)
)

convert_xlsx_to_source(
  file,
  importArgs = list(),
  omit_empty_rows = TRUE,
  output = NULL,
cols_to_utterances = NULL,
cols_to_ciids = NULL,
cols_to_codes = NULL,
cols_to_attributes = NULL,
oneFile = TRUE,
cols_to_sourceFilename = cols_to_ciids,
cols_in_sourceFilename_sep = "=",
sourceFilename_prefix = "source_",
sourceFilename_suffix = "",
Ciid_labels = NULL,
Ciid_separator = "=",
attributesFile = NULL,
preventOverwriting = rock::opts$getattr(preventOverwriting),
encoding = rock::opts$getattr(encoding),
silent = rock::opts$getattr(silent)
)

convert_sav_to_source(
  file,
  importArgs = NULL,
  omit_empty_rows = TRUE,
  output = NULL,
cols_to_utterances = NULL,
cols_to_ciids = NULL,
...
convert_df_to_source

```r
cols_to_codes = NULL,
cols_to_attributes = NULL,
oneFile = TRUE,
cols_to_sourceFilename = cols_to_ciids,
cols_in_sourceFilename_sep = "=",
sourceFilename_prefix = "source_",
sourceFilename_suffix = "",
ciid_labels = NULL,
ciid_separator = "=",
attributesFile = NULL,
preventOverwriting = rock::opts$\text{get}(\text{preventOverwriting}),
encoding = rock::opts$\text{get}(\text{encoding}),
silent = rock::opts$\text{get}(\text{silent})
```

**Arguments**

- **data**
  The data frame containing the data to convert.

- **output**
  If `oneFile=TRUE` (the default), the name (and path) of the file in which to save the processed source (if it is `NULL`, the resulting character vector will be returned visibly instead of invisibly). Note that the ROCK convention is to use `.rock` as extension. If `oneFile=FALSE`, the path to which to write the sources (if it is `NULL`, as a result a list of character vectors will be returned visibly instead of invisibly).

- **omit_empty_rows**
  Whether to omit rows where the values in the columns specified to convert to utterances are all empty (or contain only whitespace).

- **cols_to_utterances**
  The names of the columns to convert to utterances, as a character vector.

- **cols_to_ciids**
  The names of the columns to convert to class instance identifiers (e.g. case identifiers), as a named character vector, with the values being the column names in the data frame, and the names being the class instance identifiers (e.g. "sourceId", "fieldId", "caseId", etc).

- **cols_to_codes**
  The names of the columns to convert to codes (i.e. codes appended to every utterance), as a character vector. When writing codes, it is not possible to also write multiple utterance columns (i.e. `utterance_classId` must be `NULL`).

- **cols_to_attributes**
  The names of the columns to convert to attributes, as a named character vector, where each name is the name of the class instance identifier to attach the attribute to. If only one column is passed in `cols_to_ciids`, names can be omitted and a regular unnamed character vector can be passed.

- **utterance_classId**
  When specifying multiple columns with utterances, and `utterance_classId` is not `NULL`, the column names are considered to be class instance identifiers, and specified above each utterance using the class identifier specified here (e.g. "utterance_classId="originalColName"" yields something like \('[\text{originalColName}=\text{colName}_1]''\).
above all utterances from the column named `colName_1`). When writing multiple utterance columns, it is not possible to also write codes (i.e. `cols_to_codes` must be NULL).

**oneFile**
Whether to store everything in one source, or create one source for each row of the data (if this is set to `FALSE`, make sure that `cols_to_sourceFilename` specifies one or more columns that together uniquely identify each row; also, in that case, `output` must be an existing directory on your PC).

**cols_to_sourceFilename**
The columns to use as unique part of the filename of each source. These will be concatenated using `cols_in_sourceFilename_sep` as a separator. Note that the final string must be unique for each row in the dataset, otherwise the filenames for multiple rows will be the same and will be overwritten! By default, the columns specified with class instance identifiers are used.

**cols_in_sourceFilename_sep**
The separator to use when concatenating the `cols_to_sourceFilename`.

**sourceFilename_prefix, sourceFilename_suffix**
Strings that are prepended and appended to the `col_to_sourceFilename` to create the full filenames. Note that `.rock` will always be added to the end as extension.

**ciid_labels**
The labels for the class instance identifiers. Class instance identifiers have brief codes used in coding (e.g. `cid` is the default for Case Identifiers, often used to identify participants) as well as more ‘readable’ labels that are used in the attributes (e.g. `caselId` is the default class instance identifier for Case Identifiers). These can be specified here as a named vector, with each element being the label and the element’s name the identifier.

**ciid_separator**
The separator for the class instance identifier - by default, either an equals sign (=) or a colon (:) are supported, but an equals sign is less ambiguous.

**attributesFile**
Optionally, a file to write the attributes to if you don’t want them to be written to the source file(s).

**preventOverwriting**
Whether to prevent overwriting of output files.

**encoding**
The encoding of the source(s).

**silent**
Whether to suppress the warning about not editing the cleaned source.

**file**
The path to a file containing the data to convert.

**importArgs**
Optionally, a list with named elements representing arguments to pass when importing the file.

### Value
A source as a character vector.

### Examples
```r
### Get path to example files
eexamplePath <-
  system.file("extdata", package="rock");
```
### Get a path to file with example data frame
def exampleFile <-
  file.path(examplePath, "spreadsheet-import-test.csv");

### Read data into a data frame
def dat <-
  read.csv(exampleFile);

### Convert data frame to a source
def source_from_df <-
  convert_df_to_source(
    dat,
    cols_to_utterances = c("open_question_1", 
      "open_question_2"),
    cols_to_ciids = c(cid = "id"),
    cols_to_attributes = c("age", "gender"),
    cols_to_codes = c("code_1", "code_2"),
    cid_labels = c(cid = "caseId")
  );

### Show the result
cat(
  source_from_df,
  sep = "\n"
);

---

**create_codingScheme**  
Create a coding scheme

**Description**

This function can be used to specify a coding scheme that can then be used in analysis.

**Usage**

```r
create_codingScheme(
  id, 
  label, 
  codes, 
  codingInstructions = NULL, 
  description = "", 
  source = ""
)
```

codingScheme_peterson
codingScheme_levine
codingScheme_willis
create_cooccurrence_matrix

Arguments

id
An identifier for this coding scheme, consisting only of letters, numbers, and underscores (and not starting with a number).

label
A short human-readable label for the coding scheme.

codes
A character vector with the codes in this scheme.

codingInstructions
Coding instructions; a named character vector, where each element is a code’s coding instruction, and each element’s name is the corresponding code.

description
A description of this coding scheme (i.e. for information that does not fit in the label).

source
Optionally, a description, reference, or URL of a source for this coding scheme.

Format

An object of class rock_codingScheme of length 5.
An object of class rock_codingScheme of length 5.
An object of class rock_codingScheme of length 5.

Details

A number of coding schemes for cognitive interviews are provided:

- **codingScheme_peterson** Coding scheme from Peterson, Peterson & Powell, 2017
- **codingScheme_levine** Coding scheme from Levine, Fowler & Brown, 2005
- **codingScheme_willis** Coding scheme from Willis, 1999

Value

The coding scheme object.

create_cooccurrence_matrix

Create a co-occurrence matrix

Description

This function creates a co-occurrence matrix based on one or more coded sources. Optionally, it plots a heatmap, simply by calling the \texttt{stats::heatmap()} function on that matrix.

Usage

```r
create_cooccurrence_matrix(
  x,
  codes = x$convenience$codingLeaves,
  plotHeatmap = FALSE
)
```
Arguments

- **x**: The parsed source(s) as provided by `rock::parse_source` or `rock::parse_sources`.
- **codes**: The codes to include; by default, takes all codes.
- **plotHeatmap**: Whether to plot the heatmap.

Value

The co-occurrence matrix; a matrix.

Examples

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Parse a selection of example sources in that directory
parsedExamples <-
  rock::parse_sources(
    examplePath,
    regex = "(test|example)(.txt|.rock)"
  );

### Create cooccurrence matrix
rock::create_cooccurrence_matrix(parsedExamples);
```

---

**css**: Create HTML fragment with CSS styling

Description

Create HTML fragment with CSS styling

Usage

```r
css(
  template = "default",
  includeBootstrap = rock::opts$get("includeBootstrap")
)
```

Arguments

- **template**: The template to load; either the name of one of the ROCK templates (currently, only 'default' is available), or the path and filename of a CSS file.
- **includeBootstrap**: Whether to include the default bootstrap CSS.
Value

A character vector with the HTML fragment.

---

**`doc_to_txt`**

*Convert a document (.docx, .pdf, .odt, .rtf, or .html) to a plain text file*

---

**Description**

This used to be a thin wrapper around `textreadr::read_document()` that also writes the result to `output`, doing its best to correctly write UTF-8 (based on the approach recommended in [this blog post](#)). However, `textreadr` was archived from CRAN. It now directly wraps the functions that `textreadr` wraps: `pdftools::pdf_text()`, `striprtf::read_rtf`, and it uses `xml2` to import `.docx` and `.odt` files, and `rvest` to import `.html` files, using the code from the `textreadr` package.

**Usage**

```r
doc_to_txt(
  input,
  output = NULL,
  encoding = rock::opts$get("encoding"),
  newExt = NULL,
  preventOverwriting = rock::opts$get("preventOverwriting"),
  silent = rock::opts$get("silent")
)
```

**Arguments**

- **input**: The path to the input file.
- **output**: The path and filename to write to. If this is a path to an existing directory (without a filename specified), the input filename will be used, and the extension will be replaced with extension.
- **encoding**: The encoding to use when writing the text file.
- **newExt**: The extension to append: only used if `output = NULL` and `newExt` is not `NULL`, in which case the output will be written to a file with the same name as `input` but with `newExt` as extension.
- **preventOverwriting**: Whether to prevent overwriting existing files.
- **silent**: Whether to the silent or chatty.

**Value**

The converted source, as a character vector.
### Examples

```r
if (requireNamespace("xml2", quietly = TRUE)) {
  print(
    rock::doc_to_txt(    
      input = system.file("extdata/doc-to-test.docx", package="rock")
    )
  );
}
```

An very rudimentary example codebook specification

---

**Description**

This is a simple and relatively short codebook specification.

**Usage**

```r
eexampleCodebook_1
```

---

**Format**

An example of a codebook specification

---

**expand_attributes**

Expand categorical attribute variables to a series of dichotomous variables

**Description**

Expand categorical attribute variables to a series of dichotomous variables

**Usage**

```r
expand_attributes(
  data,
  attributes,
  valueLabels = NULL,
  prefix = "",
  glue = "--",
  suffix = "",
  falseValue = 0,
  trueValue = 1,
  valueFirst = TRUE,
  append = TRUE
)
```
Arguments

data The data frame, normally the $qdt$ data frame that exists in the object returned by a call to `parse_sources()`.

attributes The name of the attribute(s) to expand.

valueLabels It’s possible to use different names for the created variables than the values of the attributes. This can be set with the `valueLabels` argument. If only one attribute is specified, pass a named vector for `valueLabels`, and if multiple attributes are specified, pass a named list of named vectors, where the name of each vector corresponds to an attribute passed in attributes. The names of the vector elements must correspond to the values of the attributes (see the example).

prefix, suffix The prefix and suffix to add to the variables names that are returned.

glue The glue to paste the first part and the second part of the composite variable name together.

falseValue, trueValue The values to set for rows that, respectively, do not match and do match an attribute value.

valueFirst Whether to insert the attribute value first, or the attribute name, in the composite variable names.

append Whether to append the columns to the supplied data frame or not.

Value

A data.frame

Examples

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <-
  file.path(examplePath, "example-1.rock");

### Parse single example source
parsedExample <- rock::parse_source(exampleFile);

### Create a categorical attribute column
parsedExample$qdt$age_group <-
  c(rep(c("<18", "18-30", "31-60", ">60"),
  each=19),
  rep(c("<18", ">60"),
  time = c(3, 4)));

### Expand to four logical columns
parsedExample$qdt <-
  rock::expand_attributes(
  parsedExample$qdt,
```

```r
```
"age_group",
valueLabels =
c("<18" = "youngest",
"18-30" = "youngish",
"31-60" = "oldish",
">60" = "oldest"
),
valueFirst = FALSE
);

### Show some of the result

```r
table(parsedExample$qdt$age_group,
parsedExample$qdt$age_group__youngest);
table(parsedExample$qdt$age_group,
parsedExample$qdt$age_group__oldish);
```

---

**Description**

This function exports data frames or matrices to HTML, sending output to one or more of the console, viewer, and one or more files.

**Usage**

```r
exportToHTML(
  input,
  output = rock::opts$get("tableOutput"),
  tableOutputCSS = rock::opts$get("tableOutputCSS")
)
```

**Arguments**

- `input` Either a `data.frame`, `table`, or `matrix`, or a list with three elements: pre, input, and post. The pre and post are simply prepended and postpended to the HTML generated based on the input$'input' element.
- `output` The output: a character vector with one or more of "console" (the raw concatenated input, without conversion to HTML), "viewer", which uses the RStudio viewer if available, and one or more filenames in existing directories.
- `tableOutputCSS` The CSS to use for the HTML table.

**Value**

Invisibly, the (potentially concatenated) input as character vector.

**Examples**

```r
exportToHTML(mtcars[1:5, 1:5]);
```
export_codes_to_txt  
Export codes to a plain text file

Description
These function can be used to convert one or more parsed sources to HTML, or to convert all sources to tabbed sections in Markdown.

Usage
export_codes_to_txt(
  input,
  output = NULL,
  codeTree = "fullyMergedCodeTrees",
  codingScheme = "codes",
  regex = "\.*",
  onlyChildrenOf = NULL,
  leavesOnly = TRUE,
  includePath = TRUE,
  preventOverwriting = rock::opts$get(preventOverwriting),
  encoding = rock::opts$get(encoding),
  silent = rock::opts$get(silent)
)

Arguments
input  An object of class rock_parsedSource (as resulting from a call to parse_source) or of class rock_parsedSources (as resulting from a call to parse_sources).
output  The filename to write to.
codeTree  Codes from which code tree to export the codes. Valid options are fullyMergedCodeTrees, extendedDeductiveCodeTrees, deductiveCodeTrees, and inductiveCodeTrees.
codingScheme  With the ROCK, it’s possible to use multiple coding scheme’s in parallel. The ROCK default is called codes (using the double square brackets as code delimiters), but other delimiters can be used as well, and give a different name. Use codingScheme to specify which code tree you want to export, if you have multiple.
regex  An optional regular expression: only codes matching this regular expression will be selected.
onlyChildrenOf  A character vector of one or more regular expressions that specify codes within which to search. For example, if the code tree contains codes parent1 and parent2, and each have a number of child codes, and parent is passed as onlyChildrenOf, only the codes within parent are selected.
leavesOnly  Whether to only write the leaves (i.e. codes that don’t have children) or all codes in the code tree.
export_mergedSourceDf_to_csv

Description

Export a merged source data frame
Usage

```r
export_mergedSourceDf_to_csv(
  x,
  file,
  exportArgs = list(fileEncoding = rock::opts$get("encoding")),
  preventOverwriting = rock::opts$get("preventOverwriting"),
  silent = rock::opts$get("silent")
)
```

```r
export_mergedSourceDf_to_csv2(
  x,
  file,
  exportArgs = list(fileEncoding = rock::opts$get("encoding")),
  preventOverwriting = rock::opts$get("preventOverwriting"),
  silent = rock::opts$get("silent")
)
```

```r
export_mergedSourceDf_to_xlsx(
  x,
  file,
  exportArgs = NULL,
  preventOverwriting = rock::opts$get("preventOverwriting"),
  silent = rock::opts$get("silent")
)
```

```r
export_mergedSourceDf_to_sav(
  x,
  file,
  exportArgs = NULL,
  preventOverwriting = rock::opts$get("preventOverwriting"),
  silent = rock::opts$get("silent")
)
```

Arguments

- `x` The object with parsed sources.
- `file` The file to export to.
- `exportArgs` Optionally, arguments to pass to the function to use to export.
- `preventOverwriting` Whether to prevent overwriting if the file already exists.
- `silent` Whether to be silent or chatty.

Value

Silently, the object with parsed sources.
**export_to_html**

*Export parsed sources to HTML or Markdown*

**Description**

These functions can be used to convert one or more parsed sources to HTML, or to convert all sources to tabbed sections in Markdown.

**Usage**

```r
export_to_html(
  input,
  output = NULL,
  template = "default",
  fragment = FALSE,
  preventOverwriting = rock::opts$get(preventOverwriting),
  encoding = rock::opts$get(encoding),
  silent = rock::opts$get(silent)
)

export_to_markdown(
  input,
  heading = "Sources",
  headingLevel = 2,
  template = "default",
  silent = rock::opts$get(silent)
)
```

**Arguments**

- **input**: An object of class `rock_parsedSource` (as resulting from a call to `parse_source`) or of class `rock_parsedSources` (as resulting from a call to `parse_sources`).
- **output**: For `export_to_html`, either `NULL` to not write any files, or, if `input` is a single `rock_parsedSource`, the filename to write to, and if `input` is a `rock_parsedSources` object, the path to write to. This path will be created with a warning if it does not exist.
- **template**: The template to load; either the name of one of the ROCK templates (currently, only 'default' is available), or the path and filename of a CSS file.
- **fragment**: Whether to include the CSS and HTML tags (FALSE) or just return the fragment(s) with the source(s) (TRUE).
- **preventOverwriting**: For `export_to_html`, whether to prevent overwriting of output files.
- **encoding**: For `export_to_html`, the encoding to use when writing the exported source(s).
- **silent**: Whether to suppress messages.
- **heading, headingLevel**: For
extract_codings_by_coderId

Extract the codings by each coder using the coderId

Description

Extract the codings by each coder using the coderId

Usage

extract_codings_by_coderId(
  input,
  recursive = TRUE,
  filenameRegex = ".*",
  postponeDeductiveTreeBuilding = TRUE,
  ignoreOddDelimiters = FALSE,
  encoding = rock::opts$get(encoding),
  silent = rock::opts$get(silent)
)
form_to_rmd_template

Arguments

- **input**: The directory with the sources.
- **recursive**: Whether to also process subdirectories.
- **filenameRegex**: Only files matching this regular expression will be processed.
- **postponeDeductiveTreeBuilding**: Whether to build deductive code trees, or only store YAML fragments.
- **ignoreOddDelimiters**: Whether to throw an error when encountering an odd number of YAML delimiters.
- **encoding**: The encoding of the files to read.
- **silent**: Whether to be chatty or silent.

Value

An object with the read sources.

---

**form_to_rmd_template**

Convert a (pre)registration form to an R Markdown template

Description

This function creates an R Markdown template from a `{preregr}` (pre)registrations form specification. Pass it the URL to a Google Sheet holding the (pre)registration form specification (in `{preregr}` format), see the "Creating a form from a spreadsheet" vignette), the path to a file with a spreadsheet holding such a specification, or a loaded or imported `{preregr}` (pre)registration form.

Usage

```r
form_to_rmd_template(
  x,
  file = NULL,
  title = NULL,
  author = NULL,
  date = "r format(Sys.time(), "%H:%M:%S on %Y-%m-%d %Z (UTC%z)\")\",
  output = "html_document",
  yaml = list(title = title, author = author, date = date, output = output),
  includeYAML = TRUE,
  chunkOpts = "echo=FALSE, results='hide'",
  justify = FALSE,
  headingLevel = 1,
  showSpecification = FALSE,
  preventOverwriting = rock::opts$get("preventOverwriting"),
  silent = rock::opts$get("silent")
)
```
Arguments

- **x**: The (pre)registration form (as produced by a call to `preregr::form_create()` or `preregr::import_from_html()`) or initialized `preregr` object (as produced by a call to `preregr::prereg_initialize()` or `preregr::import_from_html()`); or, for the printing method, the R Markdown template produced by a call to `preregr::form_to_rmd_template()`.

- **file**: Optionally, a file to save the html to.

- **title**: The title to specify in the template’s YAML front matter.

- **author**: The author to specify in the template’s YAML front matter.

- **date**: The date to specify in the template’s YAML front matter.

- **output**: The output format to specify in the template’s YAML front matter.

- **yaml**: It is also possible to specify the YAML front matter directly using this argument. If used, it overrides anything specified in title, author, date and output.

- **includeYAML**: Whether to include the YAML front matter or omit it.

- **chunkOpts**: The chunk options to set for the chunks in the template.

- **justify**: Whether to use `preregr::prereg_specify()` as function for specifying the (pre)registration content (if FALSE), or `preregr::prereg_justify()` (if TRUE).

- **headingLevel**: The level of the top-most heading to use (the title of the (pre)registration form).

- **showSpecification**: Whether to show the specification in the Rmd output. When FALSE, the prereg option silent is set to TRUE at the start of the Rmd template; otherwise, it is set to FALSE.

- **preventOverwriting**: Set to FALSE to override overwrite prevention.

- **silent**: Whether to be silent or chatty.

Value

- **x**, invisibly

Examples

```r
preregr::form_create(
  title = "Example form",
  version = "0.1.0"
) |> 
preregr::form_to_rmd_template();
```
**generate_uids**

*Generate utterance identifiers (UIDs)*

---

**Description**

This function generates utterance identifiers.

**Usage**

```r
generate_uids(x, origin = Sys.time())
```

**Arguments**

- `x`: The number of identifiers to generate.
- `origin`: The origin to use when generating the actual identifiers. These identifiers are the present UNIX timestamp (i.e. the number of seconds elapsed since the UNIX epoch, the first of January 1970), accurate to two decimal places (i.e. to centiseconds), converted to the base 30 system using `numericToBase30()`. By default, the present time is used as origin, one one centisecond is added for every identifiers to generate. `origin` can be set to other values to work with different origins (of course, don’t use this unless you understand very well what you’re doing!).

**Value**

A vector of UIDs.

**Examples**

```r
rock::generate_uids(5);
```

```r
### Show how UIDs are the converted date/time
x <- rock::generate_uids(1);
x;
x_UID <- gsub(
   "\[[\[uid=(.*)\]\]]\]",
   "\1",
   x
);
x_as_nr <- rock::base30toNumeric(x_UID);
x_as_timestamp <- x_as_nr / 100;
x_as_date <-
as.POSIXct(
   x_as_timestamp,
   origin = "1970-01-01",
   tz = "UTC"
);
x_as_date
```
generic_recoding  

Generic underlying recoding function

Description

This function contains the general set of actions that are always used when recoding a source (e.g. check the input, document the justification, etc). Users should normally never call this function.

Usage

generic_recoding(
  input,  
codes,  
  func,  
  filenameRegex = ".*",  
  filter = TRUE,  
  output = NULL,  
  outputPrefix = "",  
  outputSuffix = ".recoded",  
  decisionLabel = NULL,  
  justification = NULL,  
  justificationFile = NULL,  
  preventOverwriting = rock::opts$get("preventOverwriting"),  
  encoding = rock::opts$get("encoding"),  
  silent = rock::opts$get("silent"),  
  ...  
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>One of 1) a character string specifying the path to a file with a source; 2) an object with a loaded source as produced by a call to <code>load_source()</code>; 3) a character string specifying the path to a directory containing one or more sources; 4) an object with a list of loaded sources as produced by a call to <code>load_sources()</code></td>
</tr>
<tr>
<td>codes</td>
<td>The codes to process</td>
</tr>
<tr>
<td>func</td>
<td>The function to apply</td>
</tr>
<tr>
<td>filenameRegex</td>
<td>Only process files matching this regular expression.</td>
</tr>
<tr>
<td>filter</td>
<td>Optionally, a filter to apply to specify a subset of the source(s) to process (see <code>get_source_filter()</code>).</td>
</tr>
<tr>
<td>output</td>
<td>If specified, the coded source will be written here.</td>
</tr>
<tr>
<td>outputPrefix, outputSuffix</td>
<td>The prefix and suffix to add to the filenames when writing the processed files to disk, in case multiple sources are passed as input.</td>
</tr>
<tr>
<td>decisionLabel</td>
<td>A description of the (recoding) decision that was taken.</td>
</tr>
<tr>
<td>justification</td>
<td>The justification for this action.</td>
</tr>
</tbody>
</table>
get_childCodeIds

justificationFile
If specified, the justification is appended to this file. If not, it is saved to the justifier::workspace(). This can then be saved or displayed at the end of the R Markdown file or R script using justifier::save_workspace().

preventOverwriting
Whether to prevent overwriting existing files when writing the files to output.

encoding
The encoding to use.

silent
Whether to be chatty or quiet.

...
Other arguments to pass to fnc.

Value
Invisibly, the recoded source(s) or source(s) object.

get_childCodeIds

Get the code identifiers a code’s descendents

Description
Get the code identifiers of all children, or all descendents (i.e. including grand-children, grand-grand-children, etc) of a code with a given identifier.

Usage
get_childCodeIds(
  x,
  parentCodeId,
  returnNodes = FALSE,
  includeParentCode = FALSE
)

get_descendentCodeIds(x, parentCodeId, includeParentCode = FALSE)

Arguments
x
 The parsed sources object

parentCodeId
 The code identifier of the parent code

returnNodes
 For get_childCodeIds(), set this to TRUE to return a list of nodes, not just the code identifiers.

includeParentCode
 Whether to include the parent code identifier in the result

Value
A character vector with code identifiers (or a list of nodes)
get_dataframe_from_nested_list

Return all values from a nested list in a dataframe

Description

Return all values from a nested list in a dataframe

Usage

get_dataframe_from_nested_list(x, nestingIn = "children")

Arguments

x
The nested list

nestingIn
The name containing the nested lists

Value

A dataframe

Examples

```
nestedList <-
list(
  id = "x",
  value = "value for x",
  children = list(
    list(
      id = "y",
      value = "value for y"
    ),
    list(
      id = "z",
      value = "value for z"
    )
  )
);
str(nestedList);
get_dataframe_from_nested_list(nestedList);
```
get_source_filter

Create a filter to select utterances in a source

Description

This function takes a character vector with regular expressions, a numeric vector with numeric indices, or a logical vector that is either as long as the source or has length 1; and then always returns a logical vector of the same length as the source.

Usage

get_source_filter(
  source,
  filter,
  ignore.case = TRUE,
  invert = FALSE,
  perl = TRUE,
  ...
)

Arguments

source  The source to produce the filter for.
filter   THe filtering criterion: a character vector with regular expressions, a numeric vector with numeric indices, or a logical vector that is either as long as the source or has length 1.
ignore.case Whether to apply the regular expression case sensitively or not (see base::grepl()).
invert   Whether to invert the result or not (i.e. whether the filter specifies what you want to select (invert=FALSE) or what you don’t want to select (invert=TRUE)).
perl     Whether the regular expression (if filter is a character vector) is a perl regular expression or not (see base::grepl()).
...      Any additional arguments are passed on to base::grepl().

Value

A logical vector of the same length as the source.
**get_state_transition_df**

Get the state transition data frame

Description
Get the state transition data frame

Usage

get_state_transition_df(x)

Arguments

x  A state transition table as produced by a call to get_state_transition_table().

Value
A dataframe with columns fromState, toState, and nrOfTransitions.

Examples

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <-
  file.path(examplePath, "state-example-1.rock");

### Parse single example source
parsedExample <- rock::parse_source(exampleFile);

### Show the state transition probabilities
exampleTable <- rock::get_state_transition_table(
  parsedExample
);

exampleStateDf <- rock::get_state_transition_df(
  exampleTable
);
```
get_state_transition_dot

*Get the state transition data frame*

**Description**
Get the state transition data frame

**Usage**
get_state_transition_dot(x)

**Arguments**
x
A state transition table as produced by a call to get_state_transition_table().

**Value**
A dataframe with columns fromState, toState, and nrOfTransitions.

**Examples**
```r
### Get path to example source
examplePath <-
    system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <-
    file.path(examplePath, "state-example-1.rock");

### Parse single example source
parsedExample <- rock::parse_source(exampleFile);

### Show the state transition probabilities
exampleTable <- rock::get_state_transition_table(parsedExample);

exampleStateDf <- rock::get_state_transition_df(exampleTable);

exampleDotCode <- rock::get_state_transition_dot(exampleStateDf);

DiagrammeR::grViz(exampleDotCode);
```
get_state_transition_table

*Get the state transition table*

**Description**

Get the state transition table

**Usage**

```r
get_state_transition_table(x, classIdentifier = "state")
```

**Arguments**

- **x**: A parsed source document as provided by `parse_source()`.
- **classIdentifier**: The identifier of the class that has the states to looks at.

**Value**

A table, with the 'from' states as rows and the 'to' states as columns

**Examples**

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <-
  file.path(examplePath, "state-example-1.rock");

### Parse single example source
parsedExample <- rock::parse_source(exampleFile);

### Show the state transition probabilities
rock::get_state_transition_table(
  parsedExample
);
```
**get_utterances_and_codes_from_source**

*Get utterances and codes from source*

**Description**

This is a convenience function to use when displaying a source. It returns an object with the raw and clean utterances in a source, as well as the utterance identifiers and a list with vectors of the codes for each utterance.

**Usage**

```
get_utterances_and_codes_from_source(x, ...)
```

**Arguments**

- **x**
  - Either the result of a call to `parse_source()`, or a path or text to pass to `parse_source()`.
- **...**
  - Arguments to `parse_source()`, which is called to parse the source.

**Value**

A list containing `utterances_raw`, `utterances_clean`, `uids`, `codeMatches`, and `codesPerUtterance`.

**Examples**

```r
### Get path to example source
exPath <- system.file("extdata", package="rock");

### Get a path to one example file
efile <- file.path(exPath, "example-1.rock");

### Parse single example source
res <- rock::get_utterances_and_codes_from_source(efile);
```
get_vectors_from_nested_list

Return one or more values from a nested list in a list of vectors

Description

Return one or more values from a nested list in a list of vectors

Usage

get_vectors_from_nested_list(x, valuesIn = NULL, nestingIn = "children")

Arguments

x The nested list
valuesIn The names holding the values to return (in vectors)
nestingIn The name containing the nested lists

Value

A list of vectors.

Examples

```r	nestedList <-
  list(
    id = "x",
    value = "value for x",
    children = list(
      list(
        id = "y",
        value = "value for y"
      ),
      list(
        id = "z",
        value = "value for z"
      )
    )
  );
str(nestedList);
get_vectors_from_nested_list(
  nestedList,
  c("id", "value")
);
```
heading

Print a heading

Description

This is just a convenience function to print a markdown or HTML heading at a given 'depth'.

Usage

```r
heading(
  ...,
  headingLevel = rock::opts$get("defaultHeadingLevel"),
  output = "markdown",
  cat = TRUE
)
```

Arguments

- `...`: The heading text: pasted together with no separator.
- `headingLevel`: The level of the heading; the default can be set with e.g. `rock::opts$set(defaultHeadingLevel=1)`.
- `output`: Whether to output to HTML ("html") or markdown (anything else).
- `cat`: Whether to cat (print) the heading or just invisibly return it.

Value

The heading, invisibly.

Examples

```r
heading("Hello ", "World", headingLevel=5);
### This produces: "\n\n##### Hello World
```

heatmap_basic

Generic convenience function to create a heatmap

Description

Generic convenience function to create a heatmap
Usage

heatmap_basic(
  data,
  x,
  y,
  fill,
  xLab = x,
  yLab = y,
  fillLab = fill,
  plotTitle = "Heatmap",
  fillScale = ggplot2::scale_fill_viridis_c(),
  theme = ggplot2::theme_minimal()
)

Arguments

data          A data frame
x, y, fill    The variables (columns) in data to use for the x axis, y axis, and fill of the heatmap, respectively.
xLab, yLab, fillLab
              The labels to use for the x axis, y axis, and fill, respectively
plotTitle    The plot title.
fillScale    The fill scale.
theme        The theme.

Value

The heatmap, as a ggplot2 object.

Examples

rock::heatmap_basic(mtcars, 'am', 'cyl', 'mpg');

inspect_coded_sources

Read sources from a directory, parse them, and show coded fragments and code tree

Description

This function combines successive calls to parse_sources(), collect_coded_fragments() and show_inductive_code_tree().
Usage

```r
inspect_coded_sources(
    path,
    parse_args = list(extension = "rock\dct", regex = NULL, recursive = TRUE,
                       ignoreOddDelimiters = FALSE, encoding = rock::opts$get("encoding"), silent =
                       rock::opts$get("silent")),
    fragments_args = list(codes = ".\*", context = 0),
    inductive_tree_args = list(codes = ".\*", output = "both", headingLevel = 3),
    deductive_tree_args = list()
)
```

Arguments

- **path**: The path containing the sources to parse and inspect.
- **parse_args**: The arguments to pass to `parse_sources()`.
- **fragments_args**: The arguments to pass to `collect_coded_fragments()`.
- **inductive_tree_args**: The arguments to pass to `show_inductive_code_tree()`.
- **deductive_tree_args**: Not yet implemented.

Value

The parsedSources object.

Examples

```r
### Get path to example source
examplePath <-
    system.file("extdata", package="rock");

### Inspect a selection of example sources - this takes too long
### to test, so hence the 'donttest' directive.

rock::inspect_coded_sources(
    examplePath,
    parse_args = list(regex = "test\(\.*\|rock\)"
)
);  
```

Description

These functions load one or more source(s) from a file or a string and store it in memory for further processing. Note that you’ll probably want to clean the sources first, using one of the `clean_sources()` functions, and you’ll probably want to add utterance identifiers to each utterance using one of the `prepending_uids()` functions.
Usage

load_source(
  input,
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent"),
  rlWarn = rock::opts$get(rlWarn),
  diligentWarnings = rock::opts$get("diligentWarnings")
)

load_sources(
  input,
  filenameRegex = ".*",
  ignoreRegex = NULL,
  recursive = TRUE,
  full.names = FALSE,
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)

Arguments

- **input**: The filename or contents of the source for `load_source` and the directory containing the sources for `load_sources`.
- **encoding**: The encoding of the file(s).
- **silent**: Whether to be chatty or quiet.
- **rlWarn**: Whether to let `readLines()` warn, e.g. if files do not end with a newline character.
- **diligentWarnings**: Whether to display very diligent warnings.
- **filenameRegex**: A regular expression to match against located files; only files matching this regular expression are processed.
- **ignoreRegex**: Regular expression indicating which files to ignore. This is a perl-style regular expression (see `base::regex`).
- **recursive**: Whether to search all subdirectories (TRUE) as well or not.
- **full.names**: Whether to store source names as filenames only or whether to include paths.

Value

Invisibly, an R character vector of classes `rock_source` and `character`.

Examples

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
```
exampleFile <-
  file.path(examplePath, "example-1.rock");

### Parse single example source
loadedSource <- rock::load_source(exampleFile);

---

**mask_source**

### Masking sources

**Description**

These functions can be used to mask a set of utterances or one or more sources.

**Usage**

```r
mask_source(
  input,
  output = NULL,
  proportionToMask = 1,
  preventOverwriting = rock::opts$get(preventOverwriting),
  encoding = rock::opts$get(encoding),
  rlWarn = rock::opts$get(rlWarn),
  maskRegex = "\[[[:alnum:]]\]",
  maskChar = "X",
  perl = TRUE,
  silent = rock::opts$get(silent)
)
```

```r
mask_sources(
  input,
  output,
  proportionToMask = 1,
  outputPrefix = "",
  outputSuffix = "_masked",
  maskRegex = "\[[[:alnum:]]\]",
  maskChar = "X",
  perl = TRUE,
  recursive = TRUE,
  filenameRegex = ".*",
  filenameReplacement = c("_PRIVATE_", "_public_"),
  preventOverwriting = rock::opts$get(preventOverwriting),
  encoding = rock::opts$get(encoding),
  silent = rock::opts$get(silent)
)
```

```r
mask_utterances(
  input,
```
proportionToMask = 1,
maskRegex = "[[[:alnum:]]]",
maskChar = "X",
perl = TRUE
)

Arguments

input    For mask_utterance, a character vector where each element is one utterance; for mask_source, either a character vector containing the text of the relevant source or a path to a file that contains the source text; for mask_sources, a path to a directory that contains the sources to mask.

output   For mask_source, if not NULL, this is the name (and path) of the file in which to save the processed source (if it is NULL, the result will be returned visibly). For mask_sources, output is mandatory and is the path to the directory where to store the processed sources. This path will be created with a warning if it does not exist. An exception is if "same" is specified - in that case, every file will be written to the same directory it was read from.

proportionToMask The proportion of utterances to mask, from 0 (none) to 1 (all).

preventOverwriting Whether to prevent overwriting of output files.

encoding The encoding of the source(s).

rlWarn Whether to let readLines() warn, e.g. if files do not end with a newline character.

maskRegex A regular expression (regex) specifying the characters to mask (i.e. replace with the masking character).

maskChar The character to replace the character to mask with.

perl Whether the regular expression is a perl regex or not.

silent Whether to suppress the warning about not editing the cleaned source.

outputPrefix, outputSuffix The prefix and suffix to add to the filenames when writing the processed files to disk.

recursive Whether to search all subdirectories (TRUE) as well or not.

filenameRegex A regular expression to match against located files; only files matching this regular expression are processed.

filenameReplacement A character vector with two elements that represent, respectively, the pattern and replacement arguments of the gsub() function. In other words, the first argument specifies a regular expression to search for in every processed filename, and the second argument specifies a regular expression that replaces any matches with the first argument. Set to NULL to not perform any replacement on the output file name.

Value

A character vector for mask_utterance and mask_source, or a list of character vectors, for mask_sources.
**match_consecutive_delimiters**

**Examples**

```r
### Mask text but not the codes
rock::mask_utterances(
  paste0(
    "Lorem ipsum dolor sit amet, consectetur adipiscing ",
    "elit. [expAttitude_expectation_73dnt5z1>earplugsFeelUnpleasant]"
  )
)
```

**Description**

This is just a convenience function that takes a vector of delimiters and returns a list of delimiter pairs.

**Usage**

```r
match_consecutive_delimiters(
  x,
  errorOnInvalidX = FALSE,
  errorOnOdd = FALSE,
  onOddIgnoreFirst = FALSE
)
```

**Arguments**

- **x** The vector with delimiter indices
- **errorOnInvalidX** Whether to return NA (if FALSE) or throw an error (if TRUE) when x is NULL, NA, or has less than 2 elements.
- **errorOnOdd** Whether to throw an error if the number of delimiter indices is odd.
- **onOddIgnoreFirst** If the number of delimiter indices is odd and no error is thrown, whether to ignore the first (TRUE) or the last (FALSE) delimiter.

**Value**

A list where each element is a two-element vector with the two consecutive delimiters.
Examples

```r
erock::match_consecutive_delimiters(
  c(1, 3, 5, 10, 19, 25, 30, 70)
);

eexampleText <- c(
  "some text",
  "delimiter",
  "more text",
  "delimiter",
  "filler text",
  "intentionally left blank",
  "delimiter",
  "final text",
  "delimiter"
);

erock::match_consecutive_delimiters(
  grep(
    "delimiter",
    exampleText
  )
);
```

merge_sources

\[\text{Merge source files by different coders}\]

Description

This function takes sets of sources and merges them using the utterance identifiers (UIDs) to match them.

Usage

```r
merge_sources(
  input,
  output,
  outputPrefix = "",
  outputSuffix = "_merged",
  primarySourcesRegex = ".*",
  primarySourcesIgnoreRegex = outputSuffix,
  primarySourcesPath = input,
  recursive = TRUE,
  primarySourcesRecursive = recursive,
  filenameRegex = ".*",
  primarySourcesFileList = NULL,
  sourcesFileList = NULL,
  postponeDeductiveTreeBuilding = TRUE,
)```
ignoreOddDelimiters = FALSE,
preventOverwriting = rock::opts$getattr(preventOverwriting),
encoding = rock::opts$getattr(encoding),
silent = rock::opts$getattr(silent),
inheritSilence = FALSE
)

**Arguments**

**input**  
The directory containing the input sources.

**output**  
The path to the directory where to store the merged sources. This path will be created with a warning if it does not exist. An exception is if "same" is specified - in that case, every file will be written to the same directory it was read from.

**outputPrefix**, **outputSuffix**  
A pre- and/or suffix to add to the filename when writing the merged sources (especially useful when writing them to the same directory).

**primarySourcesRegex**  
A regular expression that specifies how to recognize the primary sources (i.e. the files used as the basis, to which the codes from other sources are added).

**primarySourcesIgnoreRegex**  
A regular expression that specifies which files to ignore as primary files.

**primarySourcesPath**  
The path containing the primary sources.

**recursive**, **primarySourcesRecursive**  
Whether to read files from sub-directories (TRUE) or not.

**filenameRegex**  
Only files matching this regular expression are read.

**primarySourcesFileList**, **sourcesFileList**  
Alternatively to using regular expressions, lists of full paths and filenames to the primary sources and all sources to process can be specified using these arguments. If this is used, neither can be NULL.

**postponeDeductiveTreeBuilding**  
Whether to immediately try to build the deductive tree(s) based on the information in this file (FALSE) or whether to skip that. Skipping this is useful if the full tree information is distributed over multiple files (in which case you should probably call `parse_sources` instead of `parse_source`).

**ignoreOddDelimiters**  
If an odd number of YAML delimiters is encountered, whether this should result in an error (FALSE) or just be silently ignored (TRUE).

**preventOverwriting**  
Whether to prevent overwriting existing files or not.

**encoding**  
The encoding of the file to read (in file).

**silent**  
Whether to provide (FALSE) or suppress (TRUE) more detailed progress updates.

**inheritSilence**  
If not silent, whether to let functions called by `merge_sources` inherit that setting.
**Value**

Invisibly, a list of the parsed, primary, and merged sources.

---

**number_as_xl_date**  
*Convert a number to a date using Excel’s system*

---

**Description**

Convert a number to a date using Excel’s system.

**Usage**

`number_as_xl_date(x)`

**Arguments**

- `x`  
The number(s)

**Value**

The date(s)

**Examples**

```r
preregr::number_as_xl_date(44113);
```

---

**opts**  
*Options for the rock package*

---

**Description**

The `rock::opts` object contains three functions to set, get, and reset options used by the rock package. Use `rock::opts$set` to set options, `rock::opts$get` to get options, or `rock::opts$reset` to reset specific or all options to their default values.

**Usage**

`opts`

**Format**

An object of class `list` of length 4.
Details

It is normally not necessary to get or set rock options. The defaults implement the Reproducible Open Coding Kit (ROCK) standard, and deviating from these defaults therefore means the processed sources and codes are not compatible and cannot be processed by other software that implements the ROCK. Still, in some cases this degree of customization might be desirable.

The following arguments can be passed:

... For rock::opts$set, the dots can be used to specify the options to set, in the format option = value, for example, utteranceMarker = "\n". For rock::opts$reset, a list of options to be reset can be passed.

option For rock::opts$set, the name of the option to set.

default For rock::opts$get, the default value to return if the option has not been manually specified.

Some of the options that can be set (see rock::opts$defaults for the full list):

codeRegexes A named character vector with one or more regular expressions that specify how to extract the codes (that were used to code the sources). These regular expressions must each contain one capturing group to capture the codes.

idRegexes A named character vector with one or more regular expressions that specify how to extract the different types of identifiers. These regular expressions must each contain one capturing group to capture the identifiers.

sectionRegexes A named character vector with one or more regular expressions that specify how to extract the different types of sections.

autoGenerateIds The names of the idRegexes that, if missing, should receive autogenerated identifiers (which consist of 'autogenerated_' followed by an incrementing number).

noCodes This regular expression is matched with all codes after they have been extracted using the codeRegexes regular expression (i.e. they're matched against the codes themselves without, for example, the square brackets in the default code regex). Any codes matching this noCodes regular expression will be ignored, i.e., removed from the list of codes.

inductiveCodingHierarchyMarker For inductive coding, this marker is used to indicate hierarchical relationships between codes. The code at the left hand side of this marker will be considered the parent code of the code on the right hand side. More than two levels can be specified in one code (for example, if the inductiveCodingHierarchyMarker is ‘>’, the code grandparent>child>grandchild would indicate codes at three levels).

attributeContainers The name of YAML fragments containing case attributes (e.g. metadata, demographic variables, quantitative data about cases, etc).

codesContainers The name of YAML fragments containing (parts of) deductive coding trees.

delimiterRegEx The regular expression that is used to extract the YAML fragments.

codeDelimiters A character vector of two elements specifying the opening and closing delimiters of codes (conform the default ROCK convention, two square brackets). The square brackets will be escaped; other characters will not, but will be used as-is.

ignoreRegex The regular expression that is used to delete lines before any other processing. This can be used to enable adding comments to sources, which are then ignored during analysis.
includeBootstrap  Whether to include the default bootstrap CSS.

utteranceMarker  How to specify breaks between utterances in the source(s). The ROCK convention is to use a newline (\n).

coderId  A regular expression specifying the coder identifier, specified similarly to the codeRegexes.

idForOmittedCoderIds  The identifier to use for utterances that do not have a coder id (i.e. utterance that occur in a source that does not specify a coder id, or above the line where a coder id is specified).

Examples

```r
### Get the default utteranceMarker
rock::opts$get(utteranceMarker);

### Set it to a custom version, so that every line starts with a pipe
rock::opts$set(utteranceMarker = "\n|");

### Check that it worked
rock::opts$get(utteranceMarker);

### Reset this option to its default value
rock::opts$reset(utteranceMarker);

### Check that the reset worked, too
rock::opts$get(utteranceMarker);
```

parsed_sources_to_ena_network

Create an ENA network out of one or more parsed sources

Description

Create an ENA network out of one or more parsed sources

Usage

```r
parsed_sources_to_ena_network(
  x,
  unitCols,
  conversationCols = "originalSource",
  codes = x$convenience$codingLeaves,
  metadata = x$convenience$attributesVars
)
```
parsed_sources_to_ena_network

Arguments

- **x**  The parsed source(s) as provided by `rock::parse_source` or `rock::parse_sources`.
- **unitCols**  The columns that together define units (e.g. utterances in each source that belong together, for example because they’re about the same topic).
- **conversationCols**  The columns that together define conversations (e.g. separate sources, but can be something else, as well).
- **codes**  The codes to include; by default, takes all codes.
- **metadata**  The columns in the merged source dataframe that contain the metadata. By default, takes all read metadata.

Value

The result of a call to `rENA::ena.plot.network()`, if that is installed.

Examples

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Parse a selection of example sources in that directory
parsedExamples <-
  rock::parse_sources(
    examplePath,
    regex = "(test|example)(.txt|.rock)"
  );

### Add something to indicate which units belong together; normally,
### these would probably be indicated using one of the identifier,
### for example the stanza identifiers, the sid's
nChunks <- nrow(parsedExamples$mergedSourceDf) %/% 10;
parsedExamples$mergedSourceDf$units <-
  c(rep(1:nChunks, each=10), rep(max(nChunks), nrow(parsedExamples$mergedSourceDf) - (10*nChunks)));

### Generate ENA plot
enaPlot <-
  rock::parsed_sources_to_ena_network(parsedExamples,
    unitCols='units');

### Show the resulting plot
print(enaPlot);
```
Description

These functions parse one (parse_source) or more (parse_sources) sources and the contained identifiers, sections, and codes.

Usage

parse_source(
  text,
  file,
  utteranceLabelRegexes = NULL,
  ignoreOddDelimiters = FALSE,
  checkClassInstanceIds = rock::opts$get(checkClassInstanceIds),
  postponeDeductiveTreeBuilding = FALSE,
  filesWithYAML = NULL,
  removeSectionBreakRows = rock::opts$get("removeSectionBreakRows"),
  removeIdentifierRows = rock::opts$get("removeIdentifierRows"),
  removeEmptyRows = rock::opts$get("removeEmptyRows"),
  rlWarn = rock::opts$get("rlWarn"),
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)

## S3 method for class 'rock_parsedSource'
print(x, prefix = "### ", ...)

parse_sources(
  path,
  extension = "rock|dct",
  regex = NULL,
  recursive = TRUE,
  removeSectionBreakRows = rock::opts$get("removeSectionBreakRows"),
  removeIdentifierRows = rock::opts$get("removeIdentifierRows"),
  removeEmptyRows = rock::opts$get("removeEmptyRows"),
  ignoreOddDelimiters = FALSE,
  checkClassInstanceIds = rock::opts$get(checkClassInstanceIds),
  mergeInductiveTrees = FALSE,
  encoding = rock::opts$get(encoding),
  silent = rock::opts$get(silent)
)

## S3 method for class 'rock_parsedSources'
print(x, prefix = "### ", ...)
## S3 method for class 'rock_parsedSources'

plot(x, ...)

### Arguments

text, file

As text or file, you can specify a file to read with encoding encoding, which will then be read using `base::readLines()`. If the argument is named text, whether it is the path to an existing file is checked first, and if it is, that file is read. If the argument is named file, and it does not point to an existing file, an error is produced (useful if calling from other functions). A text should be a character vector where every element is a line of the original source (like provided by `base::readLines()`); although if a character vector of one element and including at least one newline character (`\n`) is provided as text, it is split at the newline characters using `base::strsplit()`. Basically, this behavior means that the first argument can be either a character vector or the path to a file; and if you’re specifying a file and you want to be certain that an error is thrown if it doesn’t exist, make sure to name it file.

utteranceLabelRegexes

Optionally, a list with two-element vectors to preprocess utterances before they are stored as labels (these `utterance perl regular expression`!

ignoreOddDelimiters

If an odd number of YAML delimiters is encountered, whether this should result in an error (FALSE) or just be silently ignored (TRUE).

checkClassInstanceIds

Whether to check for the occurrence of class instance identifiers specified in the attributes.

postponeDeductiveTreeBuilding

Whether to immediately try to build the deductive tree(s) based on the information in this file (FALSE) or whether to skip that. Skipping this is useful if the full tree information is distributed over multiple files (in which case you should probably call `parse_sources` instead of `parse_source`).

filesWithYAML

Any additional files to process to look for YAML fragments.

removeSectionBreakRows, removeIdentifierRows, removeEmptyRows

Whether to remove from the QDT, respectively: rows containing section breaks; rows containing only (class instance) identifiers; and empty rows.

rlWarn

Whether to let `readLines()` warn, e.g. if files do not end with a newline character.

encoding

The encoding of the file to read (in file).

silent

Whether to provide (FALSE) or suppress (TRUE) more detailed progress updates.

x

The object to print.

prefix

The prefix to use before the ‘headings’ of the printed result.

... Any additional arguments are passed on to the default print method.

path

The path containing the files to read.

extension

The extension of the files to read; files with other extensions will be ignored. Multiple extensions can be separated by a pipe (|).
Instead of specifying an extension, it's also possible to specify a regular expression; only files matching this regular expression are read. If specified, `regex` takes precedence over extension.

Whether to also process subdirectories (TRUE) or not (FALSE).

Merge multiple inductive code trees into one; this functionality is currently not yet implemented.

For `rock::parse_source()`, an object of class `rock_parsedSource`; for `rock::parse_sources()`, an object of class `rock_parsedSources`. These objects contain the original source(s) as well as the final data frame with utterances and codes, as well as the code structures.

### Examples

```r
### Get path to example source
examplePath <- system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <- file.path(examplePath, "example-1.rock");

### Parse single example source
parsedExample <- rock::parse_source(exampleFile);

### Show inductive code tree for the codes extracted with the regular expression specified with the name 'codes':
parsedExample$inductiveCodeTrees$codes;

### If you want 'rock' to be chatty, use:
parsedExample <- rock::parse_source(exampleFile, silent=FALSE);

### Parse as selection of example sources in that directory
parsedExamples <- rock::parse_sources(
  examplePath, 
  regex = "(test|example)(.txt|.rock)"
);

### Show combined inductive code tree for the codes extracted with the regular expression specified with the name 'codes':
parsedExamples$inductiveCodeTrees$codes;

### Show a source coded with the Qualitative Network Approach
qnaExample <- rock::parse_source(
```
parse_source_by_coderId

Parsing sources separately for each coder

Description
Parsing sources separately for each coder

Usage

parse_source_by_coderId(
    input,
    ignoreOddDelimiters = FALSE,
    postponeDeductiveTreeBuilding = TRUE,
    rlWarn = rock::opts$get(rlWarn),
    encoding = "UTF-8",
    silent = TRUE
)

parse_sources_by_coderId(
    input,
    recursive = TRUE,
    filenameRegex = ".*",
    ignoreOddDelimiters = FALSE,
    postponeDeductiveTreeBuilding = TRUE,
    encoding = rock::opts$get(encoding),
    silent = rock::opts$get(silent)
)

Arguments

input  For parse_source_by_coderId, either a character vector containing the text of the relevant source or a path to a file that contains the source text; for parse_sources_by_coderId, a path to a directory that contains the sources to parse.

ignoreOddDelimiters  If an odd number of YAML delimiters is encountered, whether this should result in an error (FALSE) or just be silently ignored (TRUE).

postponeDeductiveTreeBuilding  Whether to immediately try to build the deductive tree(s) based on the information in this file (FALSE) or whether to skip that. Skipping this is useful if the full tree
prepend_ids_to_source

Prepending unique utterance identifiers

Description

This function prepends unique utterance identifiers to each utterance (line) in a source. Note that you’ll probably want to clean the sources using `clean_sources()` first.

Usage

```r
prepend_ids_to_source(
  input,
  output = NULL,
  origin = Sys.time(),
  rlWarn = rock::opts$get(rlWarn),
  preventOverwriting = rock::opts$get(preventOverwriting),
  encoding = rock::opts$get(encoding),
)```
prepend_ids_to_source

```r
silent = rock::opts$get(silent)

prepend_ids_to_sources(
  input,
  output = NULL,
  outputPrefix = "",
  outputSuffix = "_withUIDs",
  origin = Sys.time(),
  preventOverwriting = rock::opts$get(preventOverwriting),
  encoding = rock::opts$get(encoding),
  silent = rock::opts$get(silent)
)
```

Arguments

- **input**: The filename or contents of the source for `prepend_ids_to_source` and the directory containing the sources for `prepend_ids_to_sources`.
- **output**: The filename where to write the resulting file for `prepend_ids_to_source` and the directory where to write the resulting files for `prepend_ids_to_sources`.
- **origin**: The time to use for the first identifier.
- **rlWarn**: Whether to let `readLines()` warn, e.g. if files do not end with a newline character.
- **preventOverwriting**: Whether to overwrite existing files (`FALSE`) or prevent that from happening (`TRUE`).
- **encoding**: The encoding of the file(s).
- **silent**: Whether to be chatty or quiet.
- **outputPrefix**, **outputSuffix**: The prefix and suffix to add to the filenames when writing the processed files to disk.

Value

The source with prepended uids, either invisible (if `output` if specified) or visibly (if not).

Examples

```r
### Simple example
rock::prepend_ids_to_source(
  "brief\nexample\nsource"
);

### Example including fake YAML fragments
longerExampleText <-
c(---,
  "First YAML fragment",
...)
```
prereg_initialize

Initialize a (pre)registration

Description

To initialize a (pre)registration, pass the URL to a Google Sheet holding the (pre)registration form specification (in `{preregr}` format), see the "Creating a form from a spreadsheet" vignette), the path to a file with a spreadsheet holding such a specification, or a loaded or imported `{preregr}` (pre)registration form.

Usage

```r
prereg_initialize(x, initialText = "Unspecified")
```

Arguments

- `x` The (pre)registration form specification, as a URL to a Google Sheet or online file or as the path to a locally stored file.
- `initialText` The text to initialize every field with.

Details

For an introduction to working with `{preregr}` (pre)registrations, see the "Specifying preregistration content" vignette.

Value

The empty (pre)registration specification.

Examples

```r
rock::prereg_initialize(
  "preregQE_v0_95"
);
```
print.rock_graphList

Plot the graphs in a list of graphs

Description

Plot the graphs in a list of graphs

Usage

## S3 method for class 'rock_graphList'
print(x, ...)

Arguments

x
The list of graphs

... Any other arguments are passed to DiagrammeR::render_graph().

Value

x, invisibly

qna_to_tlm

Convert a QNA network to Linear Topic Map format

Description

The Linear Topic Map format, LTM (https://ontopia.net/download/ltm.html), allows specification of networks in a human-readable format.

Usage

qna_to_tlm(
  x,
  topicmapId = "rock_qna_topicmap",
  topicmapTitle = "A ROCK QNA Topic Map"
)

Arguments

x The parsed source object (as produced by parse_source()), or an object holding multiple parsed sources (as produced by parse_sources()).

topicmapId, topicmapTitle The topic map’s identifier and title.
Value

If \( x \) is a single parsed source: a character vector holding the Linear Topic Map specification; or, if multiple network coding schemes were used in parallel, each in a list. If \( x \) contains multiple parsed sources, a list of such objects (i.e., a list of vectors, or a list of lists of vectors).

Examples

```r
### Get path to example source
eexamplePath <-
  system.file("extdata", package="rock");

### Read a source coded with the Qualitative Network Approach
defExample <-
  rock::parse_source(  
    file.path(  
      examplePath,  
      "network-example-1.rock"  
    )  
  );

### Convert and show the topic map
cat(  
  rock::qna_to_tlm(  
    qdefExample  
  ),  
  sep="\n"  
);
```

---

**rbind_dfs**

*Simple alternative for rbind.fill or bind_rows*

Description

Simple alternative for rbind.fill or bind_rows

Usage

```r
rbind_dfs(x, y, clearRowNames = TRUE)
```

Arguments

- \( x \) One dataframe
- \( y \) Another dataframe
- clearRowNames Whether to clear row names (to avoid duplication)

Value

The merged dataframe
**Examples**

```r
rbind_dfs(Orange, mtcars);
```

**Description**

Bind lots of dataframes together rowwise

**Usage**

```r
rbind_df_list(x)
```

**Arguments**

- **x**: A list of dataframes

**Value**

A dataframe

**Examples**

```r
rbind_df_list(list(Orange, mtcars, ChickWeight));
```

**read_spreadsheet**

*Convenience function to read spreadsheet-like files*

**Description**

Currently reads spreadsheets from Google Sheets or from `xlsx`, `csv`, or `sav` files.

**Usage**

```r
read_spreadsheet(
  x,
  sheet = NULL,
  columnDictionary = NULL,
  localBackup = NULL,
  exportGoogleSheet = FALSE,
  flattenSingleDf = FALSE,
  xlsxPkg = c("rw_xl", "openxlsx", "XLConnect"),
  failQuietly = FALSE,
  silent = rock::opts$get("silent")
)
```
Arguments

- **x**: The URL or path to a file.
- **sheet**: Optionally, the name(s) of the worksheet(s) to select.
- **columnDictionary**: Optionally, a dictionary with column names to check for presence. A named list of vectors.
- **localBackup**: If not NULL, a valid filename to write a local backup to.
- **exportGoogleSheet**: If `x` is a URL to a Google Sheet, instead of using the googlesheets4 package to download the data, by passing `exportGoogleSheet=TRUE`, an export link will be produced and the data will be downloaded as Excel spreadsheet.
- **flattenSingleDf**: Whether to return the result as a data frame if only one data frame is returned as a result.
- **xlsxPkg**: Which package to use to work with Excel spreadsheets.
- **failQuietly**: Whether to give an error when `x` is not a valid URL or existing file, or just return NULL invisibly.
- **silent**: Whether to be silent or chatty.

Value

A list of dataframes, or, if only one data frame was loaded and `flattenSingleDf` is TRUE, a data frame.

Examples

```r
### This requires an internet connection!
## Not run:
read_spreadsheet(
  paste0(
    "https://docs.google.com/",
    "spreadsheets/d/",
    "1bHDzpCu4CwEa5_3_q_9vh2691xPhc53e4A_jHLhw_U8"
  )
);
## End(Not run)
```

**recode_addChildCodes**  \hspace{1cm} Add child codes under a parent code

Description

This function conditionally adds new child codes under a code. Where `recode_split()` removes the original code (splitting it into the new codes), this function retains the original, adding the new codes as sub-codes.
Usage

```r
recode_addChildCodes(
  input,
  codes,
  childCodes,
  filter = TRUE,
  output = NULL,
  filenameRegex = ".*",
  outputPrefix = "",
  outputSuffix = ".rcAdded",
  decisionLabel = NULL,
  justification = NULL,
  justificationFile = NULL,
  preventOverwriting = rock::opts$get("preventOverwriting"),
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)
```

Arguments

- **input**: One of 1) a character string specifying the path to a file with a source; 2) an object with a loaded source as produced by a call to `load_source()`; 3) a character string specifying the path to a directory containing one or more sources; 4) an object with a list of loaded sources as produced by a call to `load_sources()`.
- **codes**: A single character value with the code to add the child codes to.
- **childCodes**: A named list with specifying when to add which child code. Each element of this list is a filtering criterion that will be passed on to `get_source_filter()` to create the actual filter that will be applied. The name of each element is the code that will be applied to utterances matching that filter. When calling `recode_addChildCodes()` for a single source, instead of passing the filtering criterion, it is also possible to pass a filter (i.e., the result of the call to `get_source_filter()`), which allows more finegrained control. Note that these 'child code filters' and the corresponding codes are processed sequentially in the order specified in `childCodes`. Any utterances coded with the code specified in `codes` that do not match with any of the 'child code filters' specified as the `childCodes` elements will remain unchanged. To create a catch-all ('else') category, pass ".*" or `TRUE` as a filter (see the example).
- **filter**: Optionally, a filter to apply to specify a subset of the source(s) to process (see `get_source_filter()`).
- **output**: If specified, the recoded source(s) will be written here.
- **filenameRegex**: Only process files matching this regular expression.
- **outputPrefix, outputSuffix**: The prefix and suffix to add to the filenames when writing the processed files to disk, in case multiple sources are passed as input.
- **decisionLabel**: A description of the (recoding) decision that was taken.
- **justification**: The justification for this action.
justificationFile

If specified, the justification is appended to this file. If not, it is saved to the justifier::workspace(). This can then be saved or displayed at the end of the R Markdown file or R script using justifier::save_workspace().

preventOverwriting

Whether to prevent overwriting existing files when writing the files to output.

encoding

The encoding to use.

silent

Whether to be chatty or quiet.

Value

Invisibly, the changed source(s) or source(s) object.

Examples

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <-
  file.path(examplePath, "example-1.rock");

### Load example source
loadedExampleSource <- rock::load_source(exampleFile);

### Split a code into two codes, showing progress (the backticks are used to be able to specify a name that starts with an underscore)
recode_source <-
  rock::recode_addChildCodes(
    loadedExampleSource,
    codes="childCode1",
    childCodes = list(
      `_and_` = " and ",
      `_book_` = "book",
      `_else_` = TRUE
    ),
    silent=FALSE
  );
```

---

**recode_delete**

*Remove one or more codes*

---

**Description**

These functions remove one or more codes from a source, and make it easy to justify that decision.
Usage

```r
recode_delete(
  input,
  codes,
  filter = TRUE,
  output = NULL,
  filenameRegex = ".*",
  outputPrefix = "",
  outputSuffix = "_rcDeleted",
  childrenReplaceParents = TRUE,
  recursiveDeletion = FALSE,
  decisionLabel = NULL,
  justification = NULL,
  justificationFile = NULL,
  preventOverwriting = rock::opts$get("preventOverwriting"),
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)
```

Arguments

input
One of 1) a character string specifying the path to a file with a source; 2) an object with a loaded source as produced by a call to `load_source()`; 3) a character string specifying the path to a directory containing one or more sources; 4) or an object with a list of loaded sources as produced by a call to `load_sources()`.

codes
A character vector with codes to remove.

filter
Optionally, a filter to apply to specify a subset of the source(s) to process (see `get_source_filter()`).

output
If specified, the recoded source(s) will be written here.

filenameRegex
Only process files matching this regular expression.

outputPrefix, outputSuffix
The prefix and suffix to add to the filenames when writing the processed files to disk, in case multiple sources are passed as input.

childrenReplaceParents
Whether children should be deleted (FALSE) or take their parent code’s place (TRUE). This is ignored if `recursiveDeletion=TRUE`, in which case children are always deleted.

recursiveDeletion
Whether to also delete a code’s parents (TRUE), if they have no other children, and keep doing this until the root is reached, or whether to leave parent codes alone (FALSE). This takes precedence over `childrenReplaceParents`.

decisionLabel
A description of the (recoding) decision that was taken.

justification
The justification for this action.

justificationFile
If specified, the justification is appended to this file. If not, it is saved to the `justifier::workspace()`. This can then be saved or displayed at the end of the R Markdown file or R script using `justifier::save_workspace()`.
recode_merge

Whether to prevent overwriting existing files when writing the files to output.

encoding The encoding to use.

silent Whether to be chatty or quiet.

Value

Invisibly, the recoded source(s) or source(s) object.

Examples

### Get path to example source
definedPath <-
system.file("extdata", package="rock");

### Get a path to one example file
defFile <-
file.path(examplePath, "example-1.rock");

### Load example source
loadedExample <- rock::load_source(exampleFile);

### Delete two codes, moving children to the codes' parents
recoded_source <-
rock::recode_delete(
  loadedExample,
  codes=c("childCode2", "childCode1"),
  silent=FALSE
);

### Process an entire directory
list_of_recoded_sources <-
rock::recode_delete(
  examplePath,
  codes=c("childCode2", "childCode1"),
  silent=FALSE
);

---

recode_merge \hspace{2cm} Merge two or more codes

Description

This function merges two or more codes into one.
Usage

recode_merge(
   input,
   codes,
   mergeToCode,
   filter = TRUE,
   output = NULL,
   filenameRegex = ".*",
   outputPrefix = "",
   outputSuffix = "_rcMerged",
   decisionLabel = NULL,
   justification = NULL,
   justificationFile = NULL,
   preventOverwriting = rock::opts$get("preventOverwriting"),
   encoding = rock::opts$get("encoding"),
   silent = rock::opts$get("silent")
)

Arguments

input One of 1) a character string specifying the path to a file with a source; 2) an object with a loaded source as produced by a call to load_source(); 3) a character string specifying the path to a directory containing one or more sources; 4) or an object with a list of loaded sources as produced by a call to load_sources().
codes A character vector with the codes to merge.
mergeToCode A single character vector with the merged code.
filter Optionally, a filter to apply to specify a subset of the source(s) to process (see get_source_filter()).
output If specified, the recoded source(s) will be written here.
filenameRegex Only process files matching this regular expression.
outputPrefix, outputSuffix The prefix and suffix to add to the filenames when writing the processed files to disk, in case multiple sources are passed as input.
decisionLabel A description of the (recoding) decision that was taken.
justification The justification for this action.
justificationFile If specified, the justification is appended to this file. If not, it is saved to the justifier::workspace(). This can then be saved or displayed at the end of the R Markdown file or R script using justifier::save_workspace().
preventOverwriting Whether to prevent overwriting existing files when writing the files to output.
encoding The encoding to use.
silent Whether to be chatty or quiet.
Value

Invisibly, the changed source(s) or source(s) object.

Examples

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <-
  file.path(examplePath, "example-1.rock");

### Load example source
loadedExample <- rock::load_source(exampleFile);

### Move two codes to a new parent, showing progress
recode_source <-
  rock::recode_merge(
    loadedExample,
    codes=c("childCode2", "grandchildCode2"),
    mergeToCode="mergedCode",
    silent=FALSE
  );
```

**recode_move**

Move one or more codes to a different parent

Description

These functions move a code to a different parent (and therefore, ancestry) in one or more sources.

Usage

```r
recode_move(
  input,
  codes,
  newAncestry,
  filter = TRUE,
  output = NULL,
  filenameRegex = ".*",
  outputPrefix = "",
  outputSuffix = "_rcMoved",
  decisionLabel = NULL,
  justification = NULL,
  justificationFile = NULL,
  preventOverwriting = rock::opts$get("preventOverwriting"),
  encoding = rock::opts$get("encoding")
)
```
silent = rock::opts$get("silent")
)

Arguments

- **input**: One of 1) a character string specifying the path to a file with a source; 2) an object with a loaded source as produced by a call to `load_source()`; 3) a character string specifying the path to a directory containing one or more sources; 4) or an object with a list of loaded sources as produced by a call to `load_sources()`.

- **codes**: A character vector with codes to move.

- **newAncestry**: The new parent code, optionally including the partial or full ancestry (i.e. the path of parent codes all the way up to the root).

- **filter**: Optionally, a filter to apply to specify a subset of the source(s) to process (see `get_source_filter()`).

- **output**: If specified, the recoded source(s) will be written here.

- **filenameRegex**: Only process files matching this regular expression.

- **outputPrefix, outputSuffix**: The prefix and suffix to add to the filenames when writing the processed files to disk, in case multiple sources are passed as input.

- **decisionLabel**: A description of the (recoding) decision that was taken.

- **justification**: The justification for this action.

- **justificationFile**: If specified, the justification is appended to this file. If not, it is saved to the `justifier::workspace()`. This can then be saved or displayed at the end of the R Markdown file or R script using `justifier::save_workspace()`.

- **preventOverwriting**: Whether to prevent overwriting existing files when writing the files to output.

- **encoding**: The encoding to use.

- **silent**: Whether to be chatty or quiet.

Value

Invisibly, the changed source(s) or source(s) object.

Examples

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
directory <-
  file.path(examplePath, "example-1.rock");

### Load example source
loadedExample <- rock::load_source(directory);
```
## Move two codes to a new parent, showing progress

```r
recode_source <-
  rock::recode_move(
    loadedExample,
    codes=c("childCode2", "childCode1"),
    newAncestry = "parentCode2",
    silent=FALSE
  );
```

---

### recode_rename

**Rename one or more codes**

#### Description

These functions rename one or more codes in one or more sources.

#### Usage

```r
recode_rename(
  input, # One of 1) a character string specifying the path to a file with a source; 2) an object with a loaded source as produced by a call to `load_source()`; 3) a character string specifying the path to a directory containing one or more sources; 4) or an object with a list of loaded sources as produced by a call to `load_sources()`.
  codes, # A named character vector with codes to rename. Each element should be the new code, and the element's name should be the old code (so e.g. codes = c(oldcode1 = 'newcode1', oldcode2 = 'newcode2')).
  filter = TRUE, # Optionally, a filter to apply to specify a subset of the source(s) to process (see `get_source_filter()`).
  output = NULL, # If specified, the recoded source(s) will be written here.
  filenameRegex = ".*", # Only process files matching this regular expression.
  outputPrefix = "", #
  outputSuffix = "_rcRenamed", #
  decisionLabel = NULL, #
  justification = NULL, #
  justificationFile = NULL, #
  preventOverwriting = rock::opts$get("preventOverwriting"), #
  encoding = rock::opts$get("encoding"), #
  silent = rock::opts$get("silent") #
)
```

#### Arguments

- **input**: A character string specifying the path to a file with a source; 2) an object with a loaded source as produced by a call to `load_source()`; 3) a character string specifying the path to a directory containing one or more sources; 4) or an object with a list of loaded sources as produced by a call to `load_sources()`.
- **codes**: A named character vector with codes to rename. Each element should be the new code, and the element's name should be the old code (so e.g. codes = c(oldcode1 = 'newcode1', oldcode2 = 'newcode2')).
- **filter**: Optionally, a filter to apply to specify a subset of the source(s) to process (see `get_source_filter()`).
- **output**: A character string specifying the path to a file with a source.
- **filenameRegex**: A regular expression to filter source files.

---

**Description**

These functions rename one or more codes in one or more sources.
outputPrefix, outputSuffix
The prefix and suffix to add to the filenames when writing the processed files to
disk, in case multiple sources are passed as input.
decisionLabel  A description of the (recoding) decision that was taken.
justification  The justification for this action.
justificationFile
If specified, the justification is appended to this file. If not, it is saved to the
justifier::workspace(). This can then be saved or displayed at the end of
the R Markdown file or R script using justifier::save_workspace().
preventOverwriting
Whether to prevent overwriting existing files when writing the files to output.
encoding  The encoding to use.
silent  Whether to be chatty or quiet.

Value
Invisibly, the changed source(s) or source(s) object.

Examples

```r
### Get path to example source
dirPath <-
  system.file("extdata", package="rock");

### Get a path to one example file
dirFile <-
  file.path(dirPath, "example-1.rock");

### Load example source
loadedExample <- rock::load_source(dirFile);

### Move two codes to a new parent, showing progress
recodedSource <-
  rock::recode_rename(
    loadedExample,
    codes=c(childCode2 = "grownUpCode2",
              grandchildCode2 = "almostChildCode2"),
    silent=FALSE
  );
```

---

**recode_split**  
Split a code into multiple codes

**Description**

This function conditionally splits a code into multiple codes. Note that you may want to use
recode_addChildCodes() instead to not lose the original coding.
recode_split

Usage

recode_split(
  input,
  codes,
  splitToCodes,
  filter = TRUE,
  output = NULL,
  filenameRegex = ".*",
  outputPrefix = "",
  outputSuffix = ".recoded",
  decisionLabel = NULL,
  justification = NULL,
  justificationFile = NULL,
  preventOverwriting = rock::opts$opt("preventOverwriting"),
  encoding = rock::opts$opt("encoding"),
  silent = rock::opts$opt("silent")
)

Arguments

input One of 1) a character string specifying the path to a file with a source; 2) an object with a loaded source as produced by a call to load_source(); 3) a character string specifying the path to a directory containing one or more sources; 4) or an object with a list of loaded sources as produced by a call to load_sources().

codes A single character value with the code to split.

splitToCodes A named list with specifying when to split to which new code. Each element of this list is a filtering criterion that will be passed on to get_source_filter() to create the actual filter that will be applied. The name of each element is the code that will be applied to utterances matching that filter. When calling recode_split() for a single source, instead of passing the filtering criterion, it is also possible to pass a filter (i.e. the result of the call to get_source_filter()), which allows more finegrained control. Note that these split filters and the corresponding codes are processed sequentially in the order specified in splitToCodes. This means that once an utterance that was coded with codes has been matched to one of these 'split filters' (and so, recoded with the corresponding 'split code', i.e., with the name of that split filter in splitToCodes), it will not be recoded again even if it also matches with other split filters down the line. Any utterances coded with the code to split up (i.e. specified in codes) that do not match with any of the split filters specified as the splitToCodes elements will not be recoded and so remain coded with codes. To create a catch-all ('else') category, pass ".*" or TRUE as a filter (see the example).

filter Optionally, a filter to apply to specify a subset of the source(s) to process (see get_source_filter()).

output If specified, the recoded source(s) will be written here.

filenameRegex Only process files matching this regular expression.
outputPrefix, outputSuffix
    The prefix and suffix to add to the filenames when writing the processed files to
disk, in case multiple sources are passed as input.

decisionLabel  A description of the (recoding) decision that was taken.
justification  The justification for this action.
justificationFile
    If specified, the justification is appended to this file. If not, it is saved to the
    justifier::workspace(). This can then be saved or displayed at the end of
    the R Markdown file or R script using justifier::save_workspace().
preventOverwriting
    Whether to prevent overwriting existing files when writing the files to output.
encoding  The encoding to use.
silent  Whether to be chatty or quiet.

Value
    Invisibly, the changed source(s) or source(s) object.

Examples
    ### Get path to example source
    examplePath <-
    system.file("extdata", package="rock");

    ### Get a path to one example file
    exampleFile <-
    file.path(examplePath, "example-1.rock");

    ### Load example source
    loadedExample <- rock::load_source(exampleFile);

    ### Split a code into two codes, showing progress
    recoded_source <-
    rock::recode_split(
        loadedExample,
        codes="childCode1",
        splitToCodes = list(
            and_REPLACED = " and ",
            book_REPLACED = "book",
            else_REPLACED = TRUE
        ),
        silent=FALSE
    );
repeatStr  
Repeat a string a number of times

**Description**

Repeat a string a number of times

**Usage**

```r
repeatStr(n = 1, str = " ")
```

**Arguments**

- `n`, `str`  Normally, respectively the frequency with which to repeat the string and the string to repeat; but the order of the inputs can be switched as well.

**Value**

A character vector of length 1.

**Examples**

```r
### 10 spaces:
repStr(10);

### Three euro symbols:
repStr("\u20ac", 3);
```

---

resultsOverview_allCodedFragments  
Show all coded fragments

**Description**

Show all coded fragments

**Usage**

```r
resultsOverview_allCodedFragments(
  x,
  root = "codes",
  context = 0,
  heading = NULL,
  headingLevel = 2,
  add_html_tags = TRUE,
  cleanUtterances = FALSE,
)```

resultsOverview_allCodedFragments

```r
output = NULL,
outputViewer = "viewer",
template = "default",
includeCSS = TRUE,
includeBootstrap = rock::opts$get("includeBootstrap"),
preventOverwriting = rock::opts$get(preventOverwriting),
silent = rock::opts$get(silent)
)

Arguments

x The parsed source(s) as provided by rock::parse_source or rock::parse_sources.
root The root code
context How many utterances before and after the target utterances to include in the fragments. If two values, the first is the number of utterances before, and the second, the number of utterances after the target utterances.
heading Optionally, a title to include in the output. The title will be prefixed with headingLevel hashes (#), and the codes with headingLevel+1 hashes. If NULL (the default), a heading will be generated that includes the collected codes if those are five or less. If a character value is specified, that will be used. To omit a heading, set to anything that is not NULL or a character vector (e.g. FALSE). If no heading is used, the code prefix will be headingLevel hashes, instead of headingLevel+1 hashes.
headingLevel The number of hashes to insert before the headings.
add_html_tags Whether to add HTML tags to the result.
cleanUtterances Whether to use the clean or the raw utterances when constructing the fragments (the raw versions contain all codes). Note that this should be set to FALSE to have add_html_tags be of the most use.
output Here, a path and filename can be provided where the result will be written. If provided, the result will be returned invisibly.
outputViewer If showing output, where to show the output: in the console (outputViewer='console') or in the viewer (outputViewer='viewer'), e.g. the RStudio viewer. You'll usually want the latter when outputting HTML, and otherwise the former. Set to FALSE to not output anything to the console or the viewer.
template The template to load; either the name of one of the ROCK templates (currently, only 'default' is available), or the path and filename of a CSS file.
includeCSS Whether to include the ROCK CSS in the returned HTML.
includeBootstrap Whether to include the default bootstrap CSS.
preventOverwriting Whether to prevent overwriting of output files.
silent Whether to provide (FALSE) or suppress (TRUE) more detailed progress updates.

Value

Invisibly, the coded fragments in a character vector.
Examples

```r
### Get path to example source
eexamplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
eexampleFile <-
  file.path(
    examplePath, "example-1.rock"
  );

### Parse single example source
parsedExample <-
  rock::parse_source(
    exampleFile
  );

### Show organised coded fragments in Markdown
cat(
  rock::resultsOverview_allCodedFragments(
    parsedExample
  )
);
```

---

rock
description

rock: A Reproducible Open Coding Kit

Description

This package implements an open standard for working with qualitative data, as such, it has two parts: a file format/convention and this R package that facilitates working with .rock files.

The ROCK File Format

The .rock files are plain text files where a number of conventions are used to add metadata. Normally these are the following conventions:

- The smallest 'codeable unit' is called an utterance, and utterances are separated by newline characters (i.e. every line of the file is an utterance);
- Codes are in between double square brackets: `[[code1]]` and `[[code2]]`;
- Hierarchy in inductive code trees can be indicated using the greater than sign (>): `[[parent>child1]]`;
- Utterances can have unique identifiers called 'utterance identifiers' or 'UIDs', which are unique short alphanumeric strings placed in between double square brackets after 'uid:', e.g. `[[uid:73k2q07]]`;
- Deductive code trees can be specified using YAML
The rock R Package Functions

The most important functions are `parse_source()` to parse one source and `parse_sources()` to parse multiple sources simultaneously. `clean_source()` and `clean_sources()` can be used to clean sources, and `prepend_ids_to_source()` and `prepend_ids_to_sources()` can be used to quickly generate UIDs and prepend them to each utterance in a source.

For analysis, `create_cooccurrence_matrix()`, `collapse_occurrences()`, and `collect_coded_fragments()` can be used.

---

### root_from_codePaths

Get the roots from a vector with code paths

**Description**

Get the roots from a vector with code paths

**Usage**

```r
root_from_codePaths(x)
```

**Arguments**

- `x` A vector of code paths.

**Value**

A vector with the root of each element.

**Examples**

```r
c(root_from_codePaths(
  c("codes>reason>parent_feels",
      "codes>reason>child_feels")
));
```

---

### rpe_create_source_with_items

Create a source with items to code for Response Process Evaluation

**Description**

This function creates a plain text file, a .rock source, that can be coded when conducting Response Process Evaluation.
Usage

rpe_create_source_with_items(
    data, iterationId, batchId, populationId,
    itemVarNames, metaquestionIdentifiers, metaquestionVarNames,
    itemContents, metaquestionContents, coderId,
    caseIds = NULL, outputFile = NULL,
    preventOverwriting = rock::opts$get("preventOverwriting"),
    encoding = rock::opts$get("encoding"),
    silent = rock::opts$get("silent")
)

Arguments

data A (wide) data frame containing at least the participants’ answers to the items
and to the meta questions (but optionally, the iteration, batch, and population).

iterationId, batchId, populationId
If the iteration, batch, and population identifiers are contained in the data frame
passed as data, the variable names holding that information for each participant;
otherwise, either a single value or a vector of length nrow(data) that contains
that information for each participant.

itemVarNames The variable names with the participants’ responses to the items, in a named
character vector, with each element’s name being the item’s identifier, and each
element the variable name in data holding the participants’ responses to the
item.

metaquestionIdentifiers A named list of unnamed character vectors, with each character vector element
specifying the identifier of a meta question, and each list element (i.e. the name
of each character vector) specifying the item identifier that the meta questions
in the corresponding character vector belong to.

metaquestionVarNames The variable names with the participants’ responses to the meta questions, in
a named character vector, with each element’s name being the meta question’s
identifier, and each element the variable name in data holding the participants’
responses to the meta question.

itemContents A named character vector with each item’s content, with the values being the
content and the names the item identifiers.

metaquestionContents A named character vector with each meta question’s content, with the values
being the content and the names the meta question identifiers.
save_workspace

```r
coderId The identifier of the coder that will code this source.
caseIds The variable name with the participants’ case identifiers (i.e. a unique identifier for each participant).
outputFile Optionally, a file to write the source to.
preventOverwriting Whether to overwrite existing files (FALSE) or prevent that from happening (TRUE).
encoding The encoding to use when writing the source(s).
silent Whether to be silent (TRUE) or chatty (FALSE).
```

Value

The created source, as a character vector (invisibly);

---

**save_workspace**

Save your justifications to a file

---

Description

When conducting analyses, you make many choices that ideally, you document and justify. This function saves stored justifications to a file.

Usage

```r
save_workspace(
  file = rock::opts$get("justificationFile"),
  encoding = rock::opts$get("encoding"),
  append = FALSE,
  preventOverwriting = rock::opts$get("preventOverwriting"),
  silent = rock::opts$get("silent")
)
```

Arguments

- **file** If specified, the file to export the justification to.
- **encoding** The encoding to use when writing the file.
- **append** Whether to append to the file, or replace its contents.
- **preventOverwriting** Whether to prevent overwriting an existing file.
- **silent** Whether to be silent or chatty.

Value

The result of a call to `justifier::export_justification()`.
### Examples

```r
### Get path to example source
defaultPath <-
  system.file("extdata", package="rock");

### Get a path to one example file
defaultFile <-
  file.path(defaultPath, "example-1.rock");

### Load example source
loadExample <- rock::load_source(defaultFile);

### Split a code into two codes, showing progress (the backticks are
### used to be able to specify a name that starts with an underscore)
recodedSource <-
  rock::recode_split(
    loadExample,
    codes="childCode1",
    splitToCodes = list(
      "_and_" = " and ",
      "_book_" = "book",
      "_else_" = TRUE
    ),
    silent=FALSE,
    justification = "Because this seems like a good idea"
  );

### Save this workspace to a file
temporaryFilename <- tempfile();
rock::save_workspace(file = temporaryFilename);
```

---

**show_attribute_table**  
*Show a table with all attributes in the RStudio viewer and/or console*

### Description

Show a table with all attributes in the RStudio viewer and/or console

### Usage

```r
show_attribute_table(  
  x,
  output = rock::opts$"tableOutput";
  tableOutputCSS = rock::opts$"tableOutputCSS"
)
```
Arguments

\(x\)  
A `rock_parsedSources` object (the result of a call to `rock::parse_sources`).

`output`  
The output: a character vector with one or more of "console" (the raw concatenated input, without conversion to HTML), "viewer", which uses the RStudio viewer if available, and one or more filenames in existing directories.

`tableOutputCSS`  
The CSS to use for the HTML table.

Value

\(x\), invisibly, unless being knitted into R Markdown, in which case a `knitr::asis_output()`-wrapped character vector is returned.

Examples

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <-
  file.path(examplePath, "example-1.rock");

### Load example source
loadedExample <- rock::parse_source(exampleFile);

### Show merged code tree
show_fullyMergedCodeTrees(loadedExample);
```
show_inductive_code_tree

Show the inductive code tree(s)

Description

This function shows one or more inductive code trees.

Usage

show_inductive_code_tree(
  x,
  codes = ".*",
  output = "both",
  headingLevel = 3,
  nodeStyle = list(shape = "box", fontname = "Arial"),
  edgeStyle = list(arrowhead = "none"),
  graphStyle = list(rankdir = "LR")
)

Arguments

x A rock_parsedSources object (the result of a call to rock::parse_sources).
codes A regular expression: only code trees from codes coded with a coding pattern
  with this name will be shown.
output Whether to show the code tree in the console (text), as a plot (plot), or both
  (both).
headingLevel The level of the heading to insert when showing the code tree as text.
odeStyle, edgeStyle, graphStyle  
  Arguments to pass on to, respectively, data.tree::SetNodeStyle(), data.tree::SetEdgeStyle(),
  and data.tree::SetGraphStyle().

Value

x, invisibly, unless being knitted into R Markdown, in which case a knitr::asis_output()-
  wrapped character vector is returned.
Description

This function splits long lines at a given number of characters, keeping words intact. It’s basically a wrapper around `strwrap()`.

Usage

```r
split_long_lines(
  x,
  length = 60,
  splitString = rock::opts$get("utteranceMarker")
)
```

Arguments

- **x**  
  The string (e.g. a source)
- **length**  
  The maximum length
- **splitString**  
  The character to use to split lines.

Value

A character vector.

Examples

```r
cat(
  rock::split_long_lines(
    paste0(
      "Lorem ipsum dolor sit amet, consectetur adipiscing elit. ",
      "Vestibulum et dictum urna. Donec neque nunc, lacinia vitae ",
      "varius vitae, pretium quis nibh. Aliquam pulvinar, lacus ",
      "sed varius vulputate, justo nibh blandit quam, ",
      "nec sollicitudin velit augue eget erat."
    )
  )
);
```
stripCodePathRoot  \hspace{1cm} Strip the root from a code path

Description

This function strips the root (just the first element) from a code path, using the codeTreeMarker stored in the opts object as marker.

Usage

stripCodePathRoot(x)

Arguments

x  \hspace{1cm} A vector of code paths.

Value

The modified vector of code paths.

Examples

stripCodePathRoot("codes>reason>parent_feels");

syncing_df_compress  \hspace{1cm} Compress a vector or data frame

Description

Compress a vector or data frame

Usage

syncing_df_compress(
  x,
  newLength,
  sep = " ",
  compressFun = NULL,
  compressFunPart = NULL,
  silent = rock::opts$get("silent")
)

syncing_vector_compress(
  x,
  newLength,
  sep = " ",
  compressFun = NULL,
  compressFunPart = NULL,
  silent = rock::opts$get("silent")
)
compressFun = NULL,
compressFunPart = NULL,
silent = rock::opts$get("silent")
)

Arguments

x
The vector or data frame

newLength
The new length (or number of rows for a data frame)

sep
When not specifying compressFun and compressFunPart, the paste function is used to combine elements, and in that case, sep is passed to paste as separator.

compressFun
If specified, when compressing streams, instead of pasting elements together using separator sep, the vectors are passed to function compressFun, which must accept a vector (to compress) and a single integer (with the desired resulting length of the vector).

compressFunPart
A function to apply to the segments that are automatically created; this can be passed instead of compressFun.

silent
Whether to be silent or chatty.

Value

The compressed vector or data frame

Examples

rock::syncing_vector_compress(
  1:10,
  3
);

rock::syncing_df_compress(
  mtcars[, 1:4],
  6
);

rock::syncing_df_compress(
  mtcars[, 1:4],
  6,
  compressFunPart = mean
);
syncing_df_expand  
Expand a vector or data frame

Description

Expand a vector or data frame

Usage

syncing_df_expand(
  x,
  newLength,
  fill = TRUE,
  expandFun = NULL,
  silent = rock::.opts$get("silent")
)

syncing_vector_expand(
  x,
  newLength,
  fill = TRUE,
  expandFun = NULL,
  silent = rock::.opts$get("silent")
)

Arguments

x          The vector or data frame
newLength  The new length (or number of rows for a data frame)
fill       When expanding streams, whether to duplicate elements to fill the resulting vec-
            tor. Ignored if fillFun is specified.
expandFun  If specified, when expanding streams, instead of potentially filling the new larger
            vector with elements (if fill is TRUE), the vectors are passed to function expandFun,
            which must accept a vector (to compress) and a single integer (with the desired
            resulting length of the vector).
silent     Whether to be silent or chatty.

Value

The expanded vector

Examples

rock::.syncing_vector_expand(letters[1:10], 15);
rock::.syncing_vector_expand(letters[1:10], 15, fill=FALSE);
sync_stream

Synchronize multiple streams

Description

This function maps the codes from multiple streams onto a primary stream.

Usage

sync_streams(
  x,
  primaryStream,
  columns = NULL,
  anchorsCol = rock::opts$get("anchorsCol"),
  sourceId = rock::opts$get("sourceId"),
  streamId = rock::opts$get("streamId"),
  prependStreamIdToColName = FALSE,
  appendStreamIdToColName = FALSE,
  sep = " ",
  fill = TRUE,
  compressFun = NULL,
  compressFunPart = NULL,
  expandFun = NULL,
  colNameGlue = rock::opts$get("colNameGlue"),
  silent = rock::opts$get("silent")
)

Arguments

x The object with the parsed sources.
primaryStream The identifier of the primary stream.
columns The names of the column(s) to synchronize.
anchorsCol The column containing the anchors.
sourceId The column containing the source identifiers.
streamId The column containing the stream identifiers.
prependStreamIdToColName, appendStreamIdToColName Whether to append or prepend the stream identifier before merging the dataframes together.
sep When not specifying compressFun and compressFunPart, the paste function is used to combine elements, and in that case, sep is passed to paste as separator.
fill When expanding streams, whether to duplicate elements to fill the resulting vector. Ignored if fillFun is specified.
compressFun If specified, when compressing streams, instead of pasting elements together using separator sep, the vectors are passed to function compressFun, which must accept a vector (to compress) and a single integer (with the desired resulting length of the vector).

compressFunPart A function to apply to the segments that are automatically created; this can be passed instead of compressFun.

expandFun If specified, when expanding streams, instead of potentially filling the new larger vector with elements (if fill is TRUE), the vectors are passed to function expandFun, which must accept a vector (to compress) and a single integer (with the desired resulting length of the vector).

colNameGlue When appending or prepending stream identifiers, the character(s) to use as "glue" or separator.

silent Whether to be silent (TRUE) or chatty (FALSE).

Value

The object with parsed sources, x, with the synchronization results added in the $syncResults subobject.

Examples

```R
### Get a directory with example sources
examplePath <- file.path(  
  system.file(package="rock"),  
  'extdata',  
  'streams'
);

### Parse the sources
parsedSources <- rock::parse_sources(  
  examplePath
);

### Add a dataframe, syncing all streams to primary stream !
parsedSources <- rock::sync_streams(  
  parsedSources,  
  primaryStream = "streamA",  
  columns = c("Code1", "Code2", "Code3"),  
  prependStreamIdToColName = TRUE
);

### Look at two examples
parsedSources$syncResults$mergedSourceDf[  
  ,  
  c("streamB_Code3", "streamC_Code1")
];
```
**sync_vector**    

*Sync (expand or compress) a vector*

**Description**

Sync (expand or compress) a vector

**Usage**

```r
sync_vector(
  x,  
  newLength, 
  sep = " ", 
  fill = TRUE, 
  compressFun = NULL, 
  expandFun = NULL, 
  compressFunPart = NULL, 
  silent = rock::opts$get("silent")
)
```

**Arguments**

- `x` The vector
- `newLength` The new length
- `sep` When not specifying `compressFun` and `compressFunPart`, the paste function is used to combine elements, and in that case, `sep` is passed to `paste` as separator.
- `fill` When expanding streams, whether to duplicate elements to fill the resulting vector. Ignored if `fillFun` is specified.
- `compressFun` If specified, when compressing streams, instead of pasting elements together using separator `sep`, the vectors are passed to function `compressFun`, which must accept a vector (to compress) and a single integer (with the desired resulting length of the vector).
- `expandFun` If specified, when expanding streams, instead of potentially filling the new larger vector with elements (if `fill` is TRUE), the vectors are passed to function `expandFun`, which must accept a vector (to compress) and a single integer (with the desired resulting length of the vector).
- `compressFunPart` A function to apply to the segments that are automatically created; this can be passed instead of `compressFun`.
- `silent` Whether to be silent or chatty.

**Value**

The synced vector
Examples

```r
template_ci_heatmap_1_to_pdf
rock::sync_vector(letters[1:10], 15);
rock::sync_vector(letters[1:10], 5);
```

---

**template_ci_heatmap_1_to_pdf**

Create a templated report for cognitive interviews

---

### Description

Use this function to export a templated report for cognitive interviews. To embed it in an R Markdown file, use `!!! CREATE rock::knit_codebook() !!!`

### Usage

```r
template_ci_heatmap_1_to_pdf(
  x,
  file,
  title = "Cognitive Interview: Heatmap and Coded Fragments",
  author = NULL,
  caption = "Heatmap",
  headingLevel = 1,
  silent = rock::opts$get("silent")
)
```

### Arguments

- `x` The codebook object (as produced by a call to `parse_sources()`).
- `file` The filename to save the codebook to.
- `title` The title to use.
- `author` The author to specify in the PDF.
- `caption` The caption for the heatmap.
- `headingLevel` The level of the top-most headings.
- `silent` Whether to be silent or chatty.

### Value

- `x`, invisibly
Examples

```r
### Use a temporary file to write to
tmpFile <- tempfile(fileext = ".pdf");

### Load an example CI
examplePath <- file.path(system.file(package="rock"), 'extdata');
parsedCI <- parse_source(file.path(examplePath,
"ci_example_1.rock"));

rock::template_ci_heatmap_1_to_pdf(
parsedCI,
  file = tmpFile
);
```

---

template_codebook_to_pdf

*Convert a codebook specification to PDF*

Description

Use this function to export your codebook specification to a PDF file. To embed it in an R Mark-down file, use `!!! CREATE rock::knit_codebook() !!!`

Usage

```r
template_codebook_to_pdf(
  x,
  file,
  author = NULL,
  headingLevel = 1,
  silent = rock::opts$get("silent")
)
```

Arguments

- **x**: The codebook object (as produced by a call to `codebook_fromSpreadsheet()`).
- **file**: The filename to save the codebook to.
- **author**: The author to specify in the PDF.
- **headingLevel**: The level of the top-most headings.
- **silent**: Whether to be silent or chatty.

Value

- **x**, invisibly
Examples

```r
### Use a temporary file to write to
tmpFile <- tempfile(fileext = "pdf");

### Load an example codebook
data("exampleCodebook_1", package = "rock");

rock::template_codebook_to_pdf(
  exampleCodebook_1,
  file = tmpFile
);
```

vecTxt

Easily parse a vector into a character value

Description

Easily parse a vector into a character value

Usage

```r
vecTxt(
  vector,
  delimiter = "",
  useQuote = "",
  firstDelimiter = NULL,
  lastDelimiter = " & ",
  firstElements = 0,
  lastElements = 1,
  lastHasPrecedence = TRUE
)
```

`vecTxtQ(vector, useQuote = "", ...)`

Arguments

- `vector` The vector to process.
- `delimiter`, `firstDelimiter`, `lastDelimiter` The delimiters to use for respectively the middle, first `firstElements`, and last `lastElements` elements.
- `useQuote` This character string is pre- and appended to all elements; so use this to quote all elements (`useQuote=""`), doublequote all elements (`useQuote=""`), or anything else (e.g. `useQuote=\'\`). The only difference between `vecTxt` and `vecTxtQ` is that the latter by default quotes the elements.
firstElements, lastElements
The number of elements for which to use the first respective last delimiters

lastHasPrecedence
If the vector is very short, it’s possible that the sum of firstElements and lastElements is larger than the vector length. In that case, downwardly adjust the number of elements to separate with the first delimiter (TRUE) or the number of elements to separate with the last delimiter (FALSE)?

... Any addition arguments to vecTxtQ are passed on to vecTxt.

Value
A character vector of length 1.

Examples
vecTxtQ(names(mtcars));

wrapVector
Wrap all elements in a vector

Description
Wrap all elements in a vector

Usage
wrapVector(x, width = 0.9 * getOption("width"), sep = "\n", ...)

Arguments
x The character vector
width The number of
sep The glue with which to combine the new lines
... Other arguments are passed to strwrap().

Value
A character vector
Examples

```r
res <- wrapVector(
  c(
    "This is a sentence ready for wrapping",
    "So is this one, although it's a bit longer"
  ),
  width = 10
);

print(res);
cat(res, sep="\n");
```

---

### write_source

**Write a source to a file**

#### Description

These functions write one or more source(s) from memory (as loaded by `load_source()` or `load_sources()`) to a file.

#### Usage

```r
write_source(
  x,
  output,
  encoding = rock::opts$get("encoding"),
  preventOverwriting = rock::opts$get("preventOverwriting"),
  silent = rock::opts$get("silent")
)
```

```r
write_sources(
  x,
  output,
  filenamePrefix = "",
  filenameSuffix = ".written",
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)
```

#### Arguments

- `x` The source(s).
- `output` The filename (for `rock::write_source()`) or path (for `rock::write_sources()`) to write to.
- `encoding` The encoding to use.
preventOverwriting  Whether to prevent against overwriting of the file(s) to write. Set to FALSE to overwrite.

silent  Whether to be chatty or quiet.

filenamePrefix, filenameSuffix  Optional prefixes or suffixes to pre- or append to the filenames when writing the files.

Value

In invisibly, the input (x), to enable chaining in pipes.

Examples

```r
### Get path to example source
examplePath <-
  system.file("extdata", package="rock");

### Get a path to one example file
exampleFile <-
  file.path(examplePath, "example-1.rock");

### Get a temporary file to write to
tempFile <- tempfile(fileext = ".rock")

### Pipe chain to load the example source; add a code;
### and write the result to disk
loadedSource <-
  rock::load_source(exampleFile) |>
  rock::code_source(c("Lorem Ipsum" = "lorumIpsum")) |>
  rock::write_source(tempFile);
```

---

### yaml_delimiter_indices

Get indices of YAML delimiters

Description

Get indices of YAML delimiters

Usage

yaml_delimiter_indices(x)
Arguments

x  The character vector.

Value

A numeric vector.

Examples

```r
yml_delimiter_indices(  
c("not here",  
    "---",  
    "above this one",  
    "but nothing here",  
    "below this one, too",  
    "---")  
);
```  
```r
### [1] 2 6
```
Index

* datasets
  - create_codingScheme, 35
  - exampleCodebook_1, 39
  - opts, 68

  add_html_tags, 4
  apply_graph_theme, 5

  base30conversion (base30toNumeric), 6
  base30toNumeric, 6
  base::grepl(), 53
  base::readLines(), 73
  base::regex, 62
  base::strsplit(), 73

  cat, 7
  cat0, 7
  checkPkgs, 7
  ci_get_item, 8
  ci_heatmap, 9
  ci_import_nrm_spec, 10
  ci_import_nrm_spec(), 9
  clean_source, 12
  clean_source(), 97
  clean_sources (clean_source), 12
  clean_sources(), 61, 76, 97
  cleaned_source_to_utterance_vector, 11
  code_freq_hist, 21
  code_source, 22
  code_source (code_source), 22
  codebook_fromSpreadsheets, 17
  codebook_fromSpreadsheets(), 19, 111
  codebook_to_pdf, 18
  codeIds_to_codePaths, 19
  codePaths_to_namedVector, 20
  codingScheme_levine
    - (create_codingScheme), 35
  codingScheme_peterson
    - (create_codingScheme), 35
  codingScheme_willis
    - (create_codingScheme), 35
  codingSchemes_get_all, 24
  collapse_occurrences, 25
  collapse_occurrences(), 97
  collect_coded_fragments, 26
  collect_coded_fragments(), 60, 61, 97
  compress_with_or (compress_with_sum), 29
  compress_with_sum, 29
  convert_csv2_to_source
    - (convert_df_to_source), 30
  convert_csv_to_source
    - (convert_df_to_source), 30
  convert_df_to_source, 30
  convert_sav_to_source
    - (convert_df_to_source), 30
  convert_xlsx_to_source
    - (convert_df_to_source), 30
  convertToNumeric, 30
  create_codingScheme, 35
  create_codingScheme(), 10
  create_cooccurrence_matrix, 36
  create_cooccurrence_matrix(), 97
  css, 37

  data.tree::SetEdgeStyle(), 102
  data.tree::SetGraphStyle(), 102
  data.tree::SetNodeStyle(), 102
  DiagrammeR::DiagrammeR, 5
  DiagrammeR::render_graph(), 79, 101
  doc_to_txt, 38

  exampleCodebook_1, 39
  expand_attributes, 39
  export_codes_to_txt, 42
  export_mergedSourceDf_to_csv, 43
  export_mergedSourceDf_to_csv2
    - (export_mergedSourceDf_to_csv), 43
export_mergedSourceDf_to_sav
  (export_mergedSourceDf_to_csv), 43
export_mergedSourceDf_to_xlsx
  (export_mergedSourceDf_to_csv), 43
export_to_html, 45
export_to_markdown (export_to_html), 45
extract_codings_by_coderId, 46
form_to_rmd_template, 47
generate_uids, 49
generate_uids(), 6
generic_recoding, 50
get (opts), 68
get_childCodeIds, 51
get_dataframe_from_nested_list, 52
get_descendentCodeIds
  (get_childCodeIds), 51
get_source_filter, 53
get_source_filter(), 50, 83, 85, 87, 89, 90, 92
get_state_transition_df, 54
get_state_transition_dot, 55
get_state_transition_table, 56
get_state_transition_table(), 54, 55
get_utterances_and_codes_from_source
  (get_childCodeIds), 51
get_vectors_from_nested_list, 58
ggplot2::ggplot(), 10, 21
gsub(), 64
heading, 59
heatmap_basic, 59
inspect_coded_sources, 60
justifier::export_justification(), 99
justifier::save_workspace(), 93
justifier::workspace(), 93
knitr::asis_output(), 101, 102
library(), 7
load_source, 61
load_source(), 50, 83, 85, 87, 89, 90, 92, 114
load_sources (load_source), 61
load_sources(), 50, 83, 85, 87, 89, 90, 92, 114
loading_sources (load_source), 61
mask_source, 63
mask_sources (mask_source), 63
mask_utterances (mask_source), 63
match_consecutive_delimiters, 65
merge_sources, 66
number_as_xl_date, 68
numericToBase30 (base30toNumeric), 6
numericToBase30(), 49
opts, 20, 68, 104
parse_source, 72
parse_source(), 9, 20, 25, 56, 57, 79, 97
parse_source_by_coderId, 75
parse_sources (parse_source), 72
parse_sources(), 9, 20, 40, 60, 61, 79, 97, 110
parse_sources_by_coderId
  (parse_source_by_coderId), 75
parsed_sources_to_ena_network, 70
parsing_sources (parse_source), 72
plot.rock_parsedSources (parse_source), 72
prepend_ids_to_source, 76
prepend_ids_to_source(), 97
prepend_ids_to_sources
  (prepend_ids_to_source), 76
prepend_ids_to_sources(), 97
prepending_uids
  (prepend_ids_to_source), 76
prepending_uids(), 61
prereg_initialize, 78
print.rock_ci_nrm (ci_import_nrm_spec), 10
print.rock_graphList, 79
print.rock_parsedSource
  (parse_source), 72
print.rock_parsedSources
  (parse_source), 72
qna_to_tlm, 79
rbind_df_list, 81
rbind_dfs, 80
read_spreadsheet, 81
INDEX

read_spreadsheet(), 11
readLines(), 14, 23, 62, 64, 73, 76, 77
recode_addChildCodes, 82
recode_addChildCodes(), 91
recode_delete, 84
recode_merge, 86
recode_move, 88
recode_rename, 90
recode_split, 91
recode_split(), 82
regex, 14, 15
repeatStr, 94
repStr (repeatStr), 94
require(), 7
reset (opts), 68
resultsOverview_allCodedFragments, 94
rock, 96
root_from_codePaths, 97
rpe_create_source_with_items, 97
save_workspace, 99
search_and_replace_in_source
  (clean_source), 12
search_and_replace_in_sources
  (clean_source), 12
set (opts), 68
show_attribute_table, 100
show_fullyMergedCodeTrees, 101
show_inductive_code_tree, 102
show_inductive_code_tree(), 60, 61
split_long_lines, 103
stats::heatmap(), 36
stripCodePathRoot, 104
strwrap(), 103, 113
sync_streams, 107
sync_vector, 109
syncing_df_compress, 104
syncing_df_expand, 106
syncing_vector_compress
  (syncing_df_compress), 104
syncing_vector_expand
  (syncing_df_expand), 106
template_ci_heatmap_1_to_pdf, 110
template_codebook_to_pdf, 111
vecTxt, 112
vecTxtQ (vecTxt), 112
wordwrap_source (clean_source), 12
wrapVector, 113
write_source, 114
write_sources (write_source), 114
writing_sources (write_source), 114
yaml_delimiter_indices, 115