

# Package ‘rock’

October 31, 2021

**Title** Reproducible Open Coding Kit

**Version** 0.5.1

**Maintainer** Gjalt-Jorn Ygram Peters <gjalt-jorn@behaviorchange.eu>

**Description** The Reproducible Open Coding Kit ('ROCK', and this package, 'rock') was developed to facilitate reproducible and open coding, specifically 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').

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

**URL** <https://r-packages.gitlab.io/rock>

**License** GPL-3

**Encoding** UTF-8

**RoxygenNote** 7.1.2

**Depends** R (>= 3.0.0)

**Imports** data.tree (>= 0.7.8), 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), purrr (>= 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, openxlsx (>= 4.2), preregr (>= 0.1.9), rENA (>= 0.1.6), readxl, rmarkdown, rstudioapi, testthat, writexl, XLConnect

**VignetteBuilder** knitr

**NeedsCompilation** no

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

**Repository** CRAN

**Date/Publication** 2021-10-31 19:40:02 UTC

## R topics documented:

add_html_tags . . . . .	3
apply_graph_theme . . . . .	4
base30toNumeric . . . . .	5
cat0 . . . . .	6
ci_get_item . . . . .	6
ci_heatmap . . . . .	7
ci_import_nrm_spec . . . . .	8
cleaned_source_to_utterance_vector . . . . .	9
clean_source . . . . .	9
codeIds_to_codePaths . . . . .	13
codePaths_to_namedVector . . . . .	14
code_freq_hist . . . . .	15
code_source . . . . .	16
codingSchemes_get_all . . . . .	18
collapse_occurrences . . . . .	19
collect_coded_fragments . . . . .	20
convert_df_to_source . . . . .	22
create_codingScheme . . . . .	27
create_cooccurrence_matrix . . . . .	28
css . . . . .	29
expand_attributes . . . . .	30
exportToHTML . . . . .	31
export_codes_to_txt . . . . .	32
export_mergedSourceDf_to_csv . . . . .	34
export_to_html . . . . .	35
extract_codings_by_coderId . . . . .	36
form_to_rmd_template . . . . .	37
generate_uids . . . . .	39
generic_recoding . . . . .	40
get_childCodeIds . . . . .	41
get_source_filter . . . . .	42
heading . . . . .	43
inspect_coded_sources . . . . .	43
load_source . . . . .	44
mask_source . . . . .	46
match_consecutive_delimiters . . . . .	48
merge_sources . . . . .	49
opts . . . . .	50
parsed_sources_to_ena_network . . . . .	52
parse_source . . . . .	53
parse_source_by_coderId . . . . .	56
prepend_ids_to_source . . . . .	57
prereg_initialize . . . . .	59
print.rock_graphList . . . . .	60
rbind_dfs . . . . .	60
rbind_df_list . . . . .	61

read_spreadsheet	61
recode_addChildCodes	63
recode_delete	65
recode_merge	66
recode_move	68
recode_rename	70
recode_split	71
repeatStr	73
rock	74
root_from_codePaths	74
save_workspace	75
show_attribute_table	76
show_inductive_code_tree	77
stripCodePathRoot	78
vecTxt	78
wrapVector	79
yaml_delimiter_indices	80

---

add_html_tags	<i>Add HTML tags to a source</i>
---------------	----------------------------------

---

## Description

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

## Usage

```
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

```
### 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 multiple DiagrammeR global graph attributes*

## Description

Apply multiple DiagrammeR global graph attributes

## Usage

```
apply_graph_theme(graph, ...)
```

## Arguments

graph	The <a href="#">DiagrammeR::DiagrammeR</a> 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

```
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**

```
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**

```
cat0(..., sep = "")
```

**Arguments**

- ... The character vector(s) to print; passed to [cat](#).
- sep The separator to pass to [cat](#), of course, "" by default.

**Value**

Nothing (invisible NULL, like [cat](#)).

**Examples**

```
cat0("The first variable is '", names(mtcars)[1], "'.")
```

---

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)
```

## Arguments

nrm_spec	The Narrative Response Model specification.
item_id	The identifier of the requested item.
language	The language of the stimuli.

## 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

```
ci_heatmap(
  x,
  itemIdentifier = "uiid",
  codingScheme = "peterson",
  itemlab = "Item",
  codelab = "Code",
  freqlab = "Frequency",
  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 <a href="#">parse_source()</a> or <a href="#">parse_sources()</a> .
itemIdentifier	The column identifying the items.
codingScheme	The coding scheme, either as a string if it represents one of the cognitive interviewing coding schemes provided with the rock package, or as a coding scheme resulting from a call to <a href="#">create_codingScheme()</a> .
itemlab, codelab, freqlab	Labels to use for the item and code axes and for the frequency color legend.
plotTitle	The title to use for the plot
fillScale	Convenient way to specify the fill scale (the colours)
theme	Convenient way to specify the <a href="#">ggplot2::ggplot()</a> theme.

**Value**

The heatmap

**Examples**

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

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

## 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 <a href="#">read_spreadsheet()</a> .
read_ss_args	A named list with arguments to pass to <a href="#">read_spreadsheet()</a> .
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**

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

**Arguments**

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

**Examples**

```
cleaned_source_to_utterance_vector("first\nsecond\nthird");
```

---

**clean\_source**

*Cleaning & editing sources*

---

**Description**

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

**Usage**

```

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")
)

```

## Arguments

input	For <code>clean_source</code> and <code>search_and_replace_in_source</code> , either a character vector containing the text of the relevant source <i>or</i> a path to a file that contains the source text; for <code>clean_sources</code> and <code>search_and_replace_in_sources</code> , a path to a directory that contains the sources to clean.
output	For <code>clean_source</code> and <code>search_and_replace_in_source</code> , if not <code>NULL</code> , this is the name (and path) of the file in which to save the processed source (if it <i>is</i> <code>NULL</code> , the result will be returned visibly). For <code>clean_sources</code> and <code>search_and_replace_in_sources</code> , <code>output</code> 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 <a href="#">regex</a> for more information). Instead of regular expressions, simple words or phrases can also be entered of course (since those are valid regular expressions). <code>replacementsPre</code> are executed before the <code>utteranceSplits</code> are applied; <code>replacementsPost</code> afterwards.
extraReplacementsPre, extraReplacementsPost	To perform more replacements than the default set, these can be conveniently specified in <code>extraReplacementsPre</code> and <code>extraReplacementsPost</code> . 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. <b>Be careful:</b> 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 <a href="#">readLines()</a> 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 <code>utteranceMarker</code> .
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 <a href="#">regex</a> for more information). Instead of regular expressions, simple words or phrases can also be entered of course (since those are valid regular expressions).

## 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, wheread 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`.

## Examples

```
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"
);

exampleSource <-
"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])", ".\\1\\U\\2"))));

```

---

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.

**Usage**

```
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**

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

---

code_freq_hist	<i>Create a frequency histogram for codes</i>
----------------	---

---

## 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\(\)](#).

---

code_source	<i>Add one or more codes to one or more sources</i>
-------------	---

---

## 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,
  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",
  recursive = TRUE,
  filenameRegex = ".*",
  preventOverwriting = rock::opts$get("preventOverwriting"),
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)
```

## Arguments

<code>input</code>	The source, or list of sources, as produced by one of the <code>loading_sources</code> functions.
<code>codes</code>	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 <code>input</code> is a source as loaded by <code>loading_sources</code> , <code>indices</code> can be used to pass a logical vector of the same length as <code>input</code> that indicates to which utterance the code in <code>codes</code> should be applied. Note that if <code>indices</code> is provided, only the first element of <code>codes</code> is used, and its name is ignored.
output	If specified, the coded source will be written here.
preventOverwriting	Whether to prevent overwriting existing files.
rlWarn	Whether to let <code>readLines()</code> 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 <code>input</code> directory
filenameRegex	Only input files matching this patterns will be read.

## Value

Invisibly, the coded source 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");

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

### Show line 71
cat(loadedExample[71]);

### Specify the rules to code all utterances
### containing "Ipsum" with the code 'ipsum' and
### all utterances containing the code
codeSpecs <-
  c("(?i)ipsum" = "ipsum",
    "BC|AD|\\d\\\\d\\\\d\\\\ds" = "timeRef");

### Apply rules
codedExample <- code_source(loadedExample,
```

```

codeSpecs);

### Show line 71
cat(codedExample[71]);

### Also add code "foo" to utterances with code 'ipsum'
moreCodedExample <- code_source(codedExample,
                                c("[[ipsum]]" = "foo"));

### Show line 71
cat(moreCodedExample[71]);

### Use the 'indices' argument to add the code 'bar' to
### line 71
overCodedExample <- code_source(moreCodedExample,
                                 "bar",
                                 indices=71);

cat(overCodedExample[71]);

```

---

`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()
```

## 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 <a href="#">parse_source()</a> .
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 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

```
### Get path to example source
exampleFile <-
  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

```

collect_coded_fragments(
  x,
  codes = ".*",
  context = 0,
  attributes = NULL,
  heading = NULL,
  headingLevel = 3,
  add_html_tags = TRUE,
  cleanUtterances = FALSE,
  output = NULL,
  outputViewer = "viewer",
  template = "default",
  rawResult = FALSE,
  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 <code>rock::parse_source</code> or <code>rock::parse_sources</code> .
codes	The regular expression that matches the codes to include

context	How many utterances before and after the target utterances to include in the fragments.
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.
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.
rawResult	Whether to return the raw result, a list of the fragments, or one character value in markdown format.
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.

## 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

```
### 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'
  )
);
```

---

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**

```
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,  
  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$get(preventOverwriting),  
  encoding = rock::opts$get(encoding),  
  silent = rock::opts$get(silent)  
)  
  
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$get(preventOverwriting),  
  encoding = rock::opts$get(encoding),  
  silent = rock::opts$get(silent)  
)  
  
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$get(preventOverwriting),
encoding = rock::opts$get(encoding),
silent = rock::opts$get(silent)
)

convert_xlsx_to_source(
  file,
  importArgs = list(overwrite = !preventOverwriting),
  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$get(preventOverwriting),
  encoding = rock::opts$get(encoding),
  silent = rock::opts$get(silent)
)

convert_sav_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$get(preventOverwriting),
  encoding = rock::opts$get(encoding),
  silent = rock::opts$get(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 vectors.
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
cols_to_codes	The names of the columns to convert to codes (i.e. codes appended to every utterance), as a character vectors.
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 unnames character vector can be passed.
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 <i>must</i> 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. 'caseId' 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, as a colon is also used for different types of codes (e.g. codes for cognitive interviews start with ci:, and unique construct identifiers (UCIDs) from psyverse start with dct:).
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

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

### Get a path to file with example data frame
exampleFile <-
  file.path(examplePath, "spreadsheet-import-test.csv");

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

### Convert data frame to a source
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"),
ciid_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

```
create_codingScheme(
```

```

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

```
codingScheme_peterson
```

```
codingScheme_levine
```

```
codingScheme_willis
```

## Arguments

<code>id</code>	An identifier for this coding scheme, consisting only of letters, numbers, and underscores (and not starting with a number).
<code>label</code>	A short human-readable label for the coding scheme.
<code>codes</code>	A character vector with the codes in this scheme.
<code>codingInstructions</code>	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 `stats:::heatmap()` function on that matrix.

## Usage

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

## Arguments

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

**Value**

The co-occurrence matrix; a `matrix`.

**Examples**

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

### Parse all example sources in that directory
parsedExamples <- rock::parse_sources(examplePath);

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

---

css

*Create HTML fragment with CSS styling*

---

**Description**

Create HTML fragment with CSS styling

**Usage**

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

**Arguments**

<code>template</code>	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.
<code>includeBootstrap</code>	Whether to include the default bootstrap CSS.

**Value**

A character vector with the HTML fragment.

---

expand_attributes	<i>Expand categorical attribute variables to a series of dichotomous variables</i>
-------------------	--

---

## Description

Expand categorical attribute variables to a series of dichotomous variables

## Usage

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

## Arguments

<b>data</b>	The data frame, normally the \$mergedSources data frame that exists in the object returned by a call to <a href="#">parse_sources()</a> .
<b>attributes</b>	The name of the attribute(s) to expand.
<b>valueLabels</b>	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).
<b>prefix, suffix</b>	The prefix and suffix to add to the variables names that are returned.
<b>glue</b>	The glue to paste the first part ad the second part of the composite variable name together.
<b>falseValue, trueValue</b>	The values to set for rows that, respectively, do not match and do match an attribute value.
<b>valueFirst</b>	Whether to insert the attribute value first, or the attribute name, in the composite variable names.
<b>append</b>	Whether to append the columns to the supplied data frame or not.

**Value**

A data.frame

**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");

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

### Create a categorical attribute column
parsedExample$mergedSourceDf$age_group <-
  rep(c("<18", "18-30", "31-60", ">60"), each=13);

### Expand to four logical columns
parsedExample$mergedSourceDf <-
  rock::expand_attributes(
    parsedExample$mergedSourceDf,
    "age_group",
    valueLabels =
      c(
        "<18" = "youngest",
        "18-30" = "youngish",
        "31-60" = "oldish",
        ">60" = "oldest"
      ),
    valueFirst = FALSE
  );

### Show result
table(parsedExample$mergedSourceDf$age_group,
      parsedExample$mergedSourceDf$age_group__youngest);
table(parsedExample$mergedSourceDf$age_group,
      parsedExample$mergedSourceDf$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**

```
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**

```
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 <code>rock_parsedSource</code> (as resulting from a call to <code>parse_source</code> ) or of class <code>rock_parsedSources</code> (as resulting from a call to <code>parse_sources</code> ).
output	The filename to write to.
codeTree	Codes from which code tree to export the codes. Valid options are <code>fullyMergedCodeTrees</code> , <code>extendedDeductiveCodeTrees</code> , <code>deductiveCodeTrees</code> , and <code>inductiveCodeTrees</code> .
codingScheme	With the ROCK, it's possible to use multiple coding scheme's in parallel. The ROCK default is called <code>codes</code> (using the double square brackets as code delimiters), but other delimiters can be used as well, and give a different name. Use <code>codingScheme</code> 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 <code>parent1</code> and <code>parent2</code> , and each have a number of child codes, and <code>parent</code> is passed as <code>onlyChildrenOf</code> , only the codes within <code>parent</code> are selected.
leavesOnly	Whether to only write the leaves (i.e. codes that don't have children) or all codes in the code tree.
includePath	Whether to only return the code itself (e.g. <code>code</code> ) or also include the path to the root (e.g. <code>code1&gt;code2&gt;code</code> ).
preventOverwriting	Whether to prevent overwriting of output files.
encoding	The encoding to use when writing the exported source(s).
silent	Whether to suppress messages.

## Value

A character vector.

## Examples

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

### Parse all example sources in that directory
parsedExamples <- rock::parse_sources(examplePath);

### Show results of exporting the codes
export_codes_to_txt(parsedExamples);

### Only show select a narrow set of codes
export_codes_to_txt(parsedExamples,
  leavesOnly=TRUE,
  includePath=FALSE,
  onlyChildrenOf = "parentCode2",
```

```
  regex="5|6");
```

---

### export\_mergedSourceDf\_to\_csv

*Export a merged source data frame*

---

## Description

Export a merged source data frame

## Usage

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

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

export_mergedSourceDf_to_xlsx(
  x,
  file,
  exportArgs = list(overwrite = !preventOverwriting),
  preventOverwriting = rock::opts$get("preventOverwriting"),
  silent = rock::opts$get("silent")
)

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	<i>Export parsed sources to HTML or Markdown</i>
----------------	--

---

**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_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 <code>rock_parsedSource</code> (as resulting from a call to <code>parse_source</code> ) or of class <code>rock_parsedSources</code> (as resulting from a call to <code>parse_sources</code> ).
-------	---

output	For <code>export_to_html</code> , either NULL to not write any files, or, if <code>input</code> is a single <code>rock_parsedSource</code> , the filename to write to, and if <code>input</code> is a <code>rock_parsedSources</code> 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 <code>export_to_html</code> , whether to prevent overwriting of output files.
encoding	For <code>export_to_html</code> , the encoding to use when writing the exported source(s).
silent	Whether to suppress messages.
heading, headingLevel	For

### Value

A character vector or a list of character vectors.

### Examples

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

### Parse all example sources in that directory
parsedExamples <- rock::parse_sources(examplePath);

### Export results to a temporary directory
tmpDir <- tempdir(check = TRUE);
prettySources <-
  export_to_html(input = parsedExamples,
                 output = tmpDir);

### Show first one
print(prettySources[[1]]);
```

---

### 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)
)
```

**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**

```
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 <code>preregr::form_create()</code> or <code>preregr::import_from_html()</code> ) or initialized <code>preregr</code> object (as produced by a call to <code>preregr::prereg_initialize()</code> or <code>preregr::import_from_html()</code> ); or, for the printing method, the R Markdown template produced by a call to <code>preregr::form_to_rmd_template()</code> .
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 <code>title</code> , <code>author</code> , <code>date</code> and <code>output</code> .
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 <code>preregr::prereg_specify()</code> as function for specifying the (pre)registration content (if FALSE), or <code>preregr::prereg_justify()</code> (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 <code>preregr</code> option <code>silent</code> 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

```
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

```
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 <a href="#">numericToBase30()</a> . By default, the present time is used as origin, 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

```
generate_uids(5);
```

---

generic_recoding	<i>Generic underlying recoding function</i>
------------------	---

---

## 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,
  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

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 <a href="#">load_source()</a> ; 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 <a href="#">load_sources()</a> .
codes	The codes to process
func	The function to apply.
filter	Optionally, a filter to apply to specify a subset of the source(s) to process (see <a href="#">get_source_filter()</a> ).
output	If specified, the coded source will be written here.
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.

**...**

Other arguments to pass to fnc.

**Value**

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

---

**get\_childCodeIds**

*Get the code identifiers of the children of a code with a given identifier*

---

**Description**

Get the code identifiers of the children of a code with a given identifier

**Usage**

```
get_childCodeIds(x, parentCodeId, returnNodes = FALSE)
```

**Arguments**

**x** The parsed sources object

**parentCodeId** The code identifier of the parent code

**returnNodes** Set to TRUE to return a list of nodes, not just the code identifiers

**Value**

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

---

get_source_filter	<i>Create a filter to select utterances in a source</i>
-------------------	---

---

## 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 <a href="#">base::grepl()</a> ).
invert	Whether to invert the result or not (i.e. whether the filter specifies what you want to select ( <code>invert=FALSE</code> ) or what you don't want to select ( <code>invert=TRUE</code> )).
perl	Whether the regular expression (if <code>filter</code> is a character vector) is a perl regular expression or not (see <a href="#">base::grepl()</a> ).
...	Any additional arguments are passed on to <a href="#">base::grepl()</a> .

## Value

A logical vector of the same length as the source.

---

heading	<i>Print a heading</i>
---------	------------------------

---

## Description

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

## Usage

```
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. <code>rock::opts\$set(defaultHeadingLevel=1)</code> .
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

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

---

inspect_coded_sources	<i>Read sources from a directory, parse them, and show coded fragments and code tree</i>
-----------------------	--

---

## Description

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

**Usage**

```
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 <a href="#">parse_sources()</a> .
fragments_args	The arguments to pass to <a href="#">collect_coded_fragments()</a> .
inductive_tree_args	The arguments to pass to <a href="#">show_inductive_code_tree()</a> .
deductive_tree_args	Not yet implemented.

**Value**

The parsedSources object.

**Examples**

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

### Inspect sources
rock::inspect_coded_sources(examplePath);
```

---

load_source	<i>Load a source from a file or a string</i>
-------------	--

---

**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 <code>load_source</code> and the directory containing the sources for <code>load_sources</code> .
encoding	The encoding of the file(s).
silent	Whether to be chatty or quiet.
rlWarn	Whether to let <code>readLines()</code> 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 <code>base::regex</code> ).
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**

```
### 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

```
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)
)

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)
)

mask_utterances(
  input,
```

```

  proportionToMask = 1,
  maskRegex = "[[:alnum:]]",
  maskChar = "X",
  perl = TRUE
)

```

## Arguments

input	For <code>mask_utterance</code> , a character vector where each element is one utterance; for <code>mask_source</code> , either a character vector containing the text of the relevant source <i>or</i> a path to a file that contains the source text; for <code>mask_sources</code> , a path to a directory that contains the sources to mask.
output	For <code>mask_source</code> , if not <code>NULL</code> , this is the name (and path) of the file in which to save the processed source (if it <i>is</i> <code>NULL</code> , the result will be returned visibly). For <code>mask_sources</code> , <code>output</code> 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 <code>readLines()</code> 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 ( <code>TRUE</code> ) 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 <code>gsub()</code> 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 <code>NULL</code> 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`.

## Examples

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

---

### match\_consecutive\_delimiters

*Match the corresponding indices of (YAML) delimiters in a sequential list*

---

## Description

Match the corresponding indices of (YAML) delimiters in a sequential list

## Usage

```
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.

---

merge_sources	<i>Merge source files by different coders</i>
---------------	---

---

## Description

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

## Usage

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

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 set-
  ting.

```

## Value

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

---

opts	<i>Options for the rock package</i>
------	-------------------------------------

---

## 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.

The following options can be set:

**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).

**persistentIds** The names of the `idRegexes` for the identifiers which, once attached to an utterance, should be attached to all following utterances as well (until a new identifier with the same name is encountered, after which that identifier will be attached to all following utterances, etc).

**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

```
### 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

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

## Arguments

x	The parsed source(s) as provided by <code>rock::parse_source</code> or <code>rock::parse_sources</code> .
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\(\)](#).

### Examples

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

### Parse all example sources in that directory
parsedExamples <- rock::parse_sources(examplePath);

### 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 function 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$get(checkClassInstanceIds),
  postponeDeductiveTreeBuilding = FALSE,
  rlWarn = rock::opts$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,
  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 <a href="#">base::readLines()</a> . 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 <a href="#">base::readLines()</a> ); 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 <a href="#">base::strsplit()</a> . 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 <code>parse_sources</code> instead of <code>parse_source</code> ).
rlWarn	Whether to let <code>readLines()</code> warn, e.g. if files do not end with a newline character.
encoding	The encoding of the file to read (in <code>file</code> ).
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 ( ).
regex	Instead of specifying an extension, it's also possible to specify a regular expression; only files matching this regular expression are read. If specified, <code>regex</code> takes precedence over <code>extension</code> ,
recursive	Whether to also process subdirectories (TRUE) or not (FALSE).
mergeInductiveTrees	Merge multiple inductive code trees into one; this functionality is currently not yet implemented.

## 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");

### 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 all example sources in that directory
parsedExamples <- rock::parse_sources(examplePath);

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

```

---

**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 <code>parse_source_by_coderId</code> , either a character vector containing the text of the relevant source <i>or</i> a path to a file that contains the source text; for <code>parse_sources_by_coderId</code> , 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 information is distributed over multiple files (in which case you should probably call <code>parse_sources</code> instead of <code>parse_source</code> ).
rlWarn	Whether to let <code>readLines()</code> warn, e.g. if files do not end with a newline character.
encoding	The encoding of the file to read (in <code>file</code> ).
silent	Whether to provide (FALSE) or suppress (TRUE) more detailed progress updates.
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.

## 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");

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

---

`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**

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

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

**Arguments**

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

```
## Simple example
```

```
rock::prepend_ids_to_source(
  "brief\nexample\nsource"
);

### Example including fake YAML fragments
longerExampleText <-
c(
  "---",
  "First YAML fragment",
  "---",
  "So this is an utterance (i.e. outside of YAML)",
  "This, too.",
  "---",
  "Second fragment",
  "---",
  "Another real utterance outside of YAML",
  "Another one outside",
  "Last 'real utterance'"
);

rock::prepend_ids_to_source(
  longerExampleText
);
```

---

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

```
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**

```
rock::prereg_initialize(
  "preregQE_v0_93"
);
```

---

`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**

<code>x</code>	The list of graphs
<code>...</code>	Any other arguments are passed to <a href="#">DiagrammeR::render_graph()</a> .

**Value**

`x`, invisibly

---

`rbind_dfs` *Simple alternative for rbind.fill or bind\_rows*

---

**Description**

Simple alternative for `rbind.fill` or `bind_rows`

**Usage**

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

**Arguments**

<code>x</code>	One dataframe
<code>y</code>	Another dataframe
<code>clearRowNames</code>	Whether to clear row names (to avoid duplication)

**Value**

The merged dataframe

**Examples**

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

---

**rbind\_df\_list**

*Bind lots of dataframes together rowwise*

---

**Description**

Bind lots of dataframes together rowwise

**Usage**

```
rbind_df_list(x)
```

**Arguments**

**x** A list of dataframes

**Value**

A dataframe

**Examples**

```
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**

```
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**

```
### This requires an internet connection!
## Not run:
read_spreadsheet(
  paste0(
    "https://docs.google.com/",
    "spreadsheets/d/",
    "1bHDzpCu4CwEa5_3_q_9vH2691XPhCS3e4Aj_HLhw_U8"
  )
)
```

```

)
);

## End(Not run)

```

---

recode\_addChildCodes *Add child codes under a parent code*

---

## 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.

## Usage

```

recode_addChildCodes(
  input,
  codes,
  childCodes,
  filter = TRUE,
  output = NULL,
  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 <a href="#">load_source()</a> ; 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 <a href="#">load_sources()</a> .
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 <a href="#">get_source_filter()</a> 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 <code>recode_addChildCodes()</code> 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 <a href="#">get_source_filter()</a> ), which allows more finegrained control. Note that these 'child code filters' and the corresponding codes are processed sequentially in the order specified in <code>childCodes</code> . Any utterances coded with the code specified in <code>codes</code> that do not match with any of the 'child code filters' specified as the <code>childCodes</code> 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 <a href="#">get_source_filter()</a> ).
output	If specified, the recoded source(s) will be written here.
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 <code>justifier::workspace()</code> . This can then be saved or displayed at the end of the R Markdown file or R script using <code>justifier::save_workspace()</code> .
preventOverwriting	Whether to prevent overwriting existing files when writing the files to <code>output</code> .
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
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)
recoded_source <-
  rock::recode_addChildCodes(
    loadedExampleSource,
    codes="childCode1",
    childCodes = list(
      `_and_` = " and ",
      `_book_` = "book",
      `_else_` = TRUE
    ),
    silent=FALSE
  );
```

---

recode_delete	<i>Remove one or more codes</i>
---------------	---------------------------------

---

## Description

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

## Usage

```
recode_delete(  
  input,  
  codes,  
  filter = TRUE,  
  output = NULL,  
  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 <a href="#">load_source()</a> ; 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 <a href="#">load_sources()</a> .
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 <a href="#">get_source_filter()</a> ).
output	If specified, the recoded source(s) will be written here.
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()`.

**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 recoded 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);

### 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
  );
```

---

<code>recode_merge</code>	<i>Merge two or more codes</i>
---------------------------	--------------------------------

---

**Description**

This function merges two or more codes into one.

**Usage**

```
recode_merge(
  input,
  codes,
  mergeToCode,
  filter = TRUE,
  output = NULL,
  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 <a href="#">load_source()</a> ; 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 <a href="#">load_sources()</a> .
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 <a href="#">get_source_filter()</a> ).
output	If specified, the recoded source(s) will be written here.
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 <a href="#">justifier::workspace()</a> . This can then be saved or displayed at the end of the R Markdown file or R script using <a href="#">justifier::save_workspace()</a> .
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);

### Move two codes to a new parent, showing progress
recoded_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

```

recode_move(
  input,
  codes,
  newAncestry,
  filter = TRUE,
  output = NULL,
  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 <a href="#">load_source()</a> ; 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 <a href="#">load_sources()</a> .
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 <a href="#">get_source_filter()</a> ).
output	If specified, the recoded source(s) will be written here.
decisionLabel	A description of the (reencoding) 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 <code>justifier::workspace()</code> . This can then be saved or displayed at the end of the R Markdown file or R script using <code>justifier::save_workspace()</code> .
preventOverwriting	Whether to prevent overwriting existing files when writing the files to <code>output</code> .
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);

### Move two codes to a new parent, showing progress
recoded_source <-
  rock::recode_move(
    loadedExample,
    codes=c("childCode2", "childCode1"),
    newAncestry = "parentCode2",
    silent=FALSE
  );
```

---

recode_rename	<i>Rename one or more codes</i>
---------------	---------------------------------

---

## Description

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

## Usage

```
recode_rename(
  input,
  codes,
  filter = TRUE,
  output = NULL,
  decisionLabel = NULL,
  justification = NULL,
  justificationFile = NULL,
  preventOverwriting = rock::opts$get("preventOverwriting"),
  encoding = rock::opts$get("encoding"),
  silent = rock::opts$get("silent")
)
```

## Arguments

<code>input</code>	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 <a href="#">load_source()</a> ; 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 <a href="#">load_sources()</a> .
<code>codes</code>	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. <code>codes = c(oldcode1 = 'newcode1', oldcode2 = 'newcode2')</code> ).
<code>filter</code>	Optionally, a filter to apply to specify a subset of the source(s) to process (see <a href="#">get_source_filter()</a> ).
<code>output</code>	If specified, the recoded source(s) will be written here.
<code>decisionLabel</code>	A description of the (recoding) decision that was taken.
<code>justification</code>	The justification for this action.
<code>justificationFile</code>	If specified, the justification is appended to this file. If not, it is saved to the <code>justifier::workspace()</code> . This can then be saved or displayed at the end of the R Markdown file or R script using <code>justifier::save_workspace()</code> .
<code>preventOverwriting</code>	Whether to prevent overwriting existing files when writing the files to <code>output</code> .
<code>encoding</code>	The encoding to use.
<code>silent</code>	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);

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

---

recode_split	<i>Split a code into multiple codes</i>
--------------	---

---

**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.

**Usage**

```
recode_split(
  input,
  codes,
  splitToCodes,
  filter = TRUE,
  output = NULL,
  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 <a href="#">load_source()</a> ; 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 <a href="#">load_sources()</a> .
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 <a href="#">get_source_filter()</a> 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 <code>recode_split()</code> 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 <a href="#">get_source_filter()</a> ), which allows more finegrained control. Note that these split filters and the corresponding codes are processed sequentially in the order specified in <code>splitToCodes</code> . This means that once an utterance that was coded with <code>codes</code> 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 <code>splitToCodes</code> ), 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 <code>codes</code> ) that do not match with any of the split filters specified as the <code>splitToCodes</code> elements will not be recoded and so remain coded with <code>codes</code> . 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 <a href="#">get_source_filter()</a> ).
output	If specified, the recoded source(s) will be written here.
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 <a href="#">justifier::workspace()</a> . This can then be saved or displayed at the end of the R Markdown file or R script using <a href="#">justifier::save_workspace()</a> .
preventOverwriting	Whether to prevent overwriting existing files when writing the files to <code>output</code> .
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 (the backticks are
### used to be able to specify a name that starts with an underscore)
recoded_source <-
  rock::recode_split(
    loadedExample,
    codes="childCode1",
    splitToCodes = list(
      `_and_` = " and ",
      `_book_` = "book",
      `_else_` = TRUE
    ),
    silent=FALSE
  );
```

---

**repeatStr**

*Repeat a string a number of times*

---

**Description**

Repeat a string a number of times

**Usage**

```
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**

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

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

rock

*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 (>): [[parent1>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:73xk2q07]];
- 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

```
root_from_codePaths(x)
```

**Arguments**

- x                   A vector of code paths.

**Value**

A vector with the root of each element.

**Examples**

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

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**

```
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

```
### 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 (the backticks are
### used to be able to specify a name that starts with an underscore)
recoded_source <-
  rock::recode_split(
    loadedExample,
    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

```
show_attribute_table(
  x,
  output = rock::opts$get("tableOutput"),
  tableOutputCSS = rock::opts$get("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.

`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.  
 nodeStyle, 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.

stripCodePathRoot *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 A vector of code paths.

**Value**

The modified vector of code paths.

**Examples**

```
stripCodePathRoot("codes>reason>parent_feels");
```

vecTxt *Easily parse a vector into a character value*

**Description**

Easily parse a vector into a character value

**Usage**

```
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 <code>firstElements</code> , and last <code>lastElements</code> elements.
useQuote	This character string is pre- and appended to all elements; so use this to quote all elements ( <code>useQuote="'"</code> ), doublequote all elements ( <code>useQuote="''"</code> ), or anything else (e.g. <code>useQuote=' '</code> ). The only difference between <code>vecTxt</code> and <code>vecTxtQ</code> is that the latter by default quotes the elements.
<code>firstElements, lastElements</code>	The number of elements for which to use the first respective last delimiters
<code>lastHasPrecedence</code>	If the vector is very short, it's possible that the sum of <code>firstElements</code> and <code>lastElements</code> 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 <code>vecTxtQ</code> are passed on to <code>vecTxt</code> .

### Value

A character vector of length 1.

### Examples

```
vecTxtQ(names(mtcars));
```

---

wrapVector	<i>Wrap all elements in a vector</i>
------------	--------------------------------------

---

### 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 <a href="#">strwrap()</a> .

**Value**

A character vector

**Examples**

```
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");
```

**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**

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

# Index

\* **datasets**  
    create\_codingScheme, 27  
    opts, 50

add\_html\_tags, 3  
apply\_graph\_theme, 4

base30conversion (base30toNumeric), 5  
base30toNumeric, 5  
base::grepl(), 42  
base::readLines(), 54  
base::regex, 45  
base::strsplit(), 54

cat, 6  
cat0, 6  
ci\_get\_item, 6  
ci\_heatmap, 7  
ci\_import\_nrm\_spec, 8  
clean\_source, 9  
clean\_source(), 74  
clean\_sources (clean\_source), 9  
clean\_sources(), 44, 57, 74  
cleaned\_source\_to\_utterance\_vector, 9  
code\_freq\_hist, 15  
code\_source, 16  
code\_sources (code\_source), 16  
codeIds\_to\_codePaths, 13  
codePaths\_to\_namedVector, 14  
codingScheme\_levine  
    (create\_codingScheme), 27  
codingScheme\_peterson  
    (create\_codingScheme), 27  
codingScheme\_willis  
    (create\_codingScheme), 27  
codingSchemes\_get\_all, 18  
collapse\_occurrences, 19  
collapse\_occurrences(), 74  
collect\_coded\_fragments, 20  
collect\_coded\_fragments(), 43, 44, 74

convert\_csv2\_to\_source  
    (convert\_df\_to\_source), 22  
convert\_csv\_to\_source  
    (convert\_df\_to\_source), 22  
convert\_df\_to\_source, 22  
convert\_sav\_to\_source  
    (convert\_df\_to\_source), 22  
convert\_xlsx\_to\_source  
    (convert\_df\_to\_source), 22  
create\_codingScheme, 27  
create\_codingScheme(), 7  
create\_cooccurrence\_matrix, 28  
create\_cooccurrence\_matrix(), 74  
css, 29

data.tree::SetEdgeStyle(), 77  
data.tree::SetGraphStyle(), 77  
data.tree::SetNodeStyle(), 77  
DiagrammeR::DiagrammeR, 4  
DiagrammeR::render\_graph(), 60

expand\_attributes, 30  
export\_codes\_to\_txt, 32  
export\_mergedSourceDf\_to\_csv, 34  
export\_mergedSourceDf\_to\_csv2  
    (export\_mergedSourceDf\_to\_csv), 34  
export\_mergedSourceDf\_to\_sav  
    (export\_mergedSourceDf\_to\_csv), 34  
export\_mergedSourceDf\_to\_xlsx  
    (export\_mergedSourceDf\_to\_csv), 34  
export\_to\_html, 35  
export\_to\_markdown (export\_to\_html), 35  
exportToHTML, 31  
extract\_codings\_by\_coderId, 36

form\_to\_rmd\_template, 37

generate\_uids, 39

generate\_uids(), 5  
 generic\_recoding, 40  
 get(opts), 50  
 get\_childCodeIds, 41  
 get\_source\_filter, 42  
 get\_source\_filter(), 40, 63–65, 67, 69, 70, 72  
 ggplot2::ggplot(), 7, 15  
 gsub(), 47  
 heading, 43  
 inspect\_coded\_sources, 43  
 justifier::export\_justification(), 75  
 justifier::save\_workspace(), 72  
 justifier::workspace(), 72  
 knitr::asis\_output(), 77, 78  
 load\_source, 44  
 load\_source(), 40, 63, 65, 67, 68, 70, 72  
 load\_sources(load\_source), 44  
 load\_sources(), 40, 63, 65, 67, 68, 70, 72  
 mask\_source, 46  
 mask\_sources(mask\_source), 46  
 mask\_utterances(mask\_source), 46  
 match\_consecutive\_delimiters, 48  
 merge\_sources, 49  
 numericToBase30(base30toNumeric), 5  
 numericToBase30(), 39  
 opts, 14, 50, 78  
 parse\_source, 53  
 parse\_source(), 7, 14, 19, 74  
 parse\_source\_by\_coderId, 56  
 parse\_sources(parse\_source), 53  
 parse\_sources(), 7, 14, 30, 43, 44, 74  
 parse\_sources\_by\_coderId  
     (parse\_source\_by\_coderId), 56  
 parsed\_sources\_to\_ena\_network, 52  
 parsing\_sources(parse\_source), 53  
 plot.rock\_parsedSources(parse\_source), 53  
 prepend\_ids\_to\_source, 57  
 prepend\_ids\_to\_source(), 74  
 prepend\_ids\_to\_sources  
     prepend\_ids\_to\_source), 57  
 prepend\_ids\_to\_sources(), 74  
 prepending\_uids  
     prepend\_ids\_to\_source), 57  
 prepending\_uids(), 44  
 prereg\_initialize, 59  
 print.rock\_ci\_nrm(ci\_import\_nrm\_spec), 8  
 print.rock\_graphList, 60  
 print.rock\_parsedSource(parse\_source), 53  
 print.rock\_parsedSources  
     (parse\_source), 53  
 rbind\_df\_list, 61  
 rbind\_dfs, 60  
 read\_spreadsheet, 61  
 read\_spreadsheet(), 8  
 readLines(), 11, 17, 45, 47, 55, 57, 58  
 recode\_addChildCodes, 63  
 recode\_addChildCodes(), 63, 71  
 recode\_delete, 65  
 recode\_merge, 66  
 recode\_move, 68  
 recode\_rename, 70  
 recode\_split, 71  
 regex, 11, 12  
 rENA::ena.plot.network(), 53  
 repeatStr, 73  
 repStr(repeatStr), 73  
 reset(opts), 50  
 rock, 74  
 root\_from\_codePaths, 74  
 save\_workspace, 75  
 search\_and\_replace\_in\_source  
     (clean\_source), 9  
 search\_and\_replace\_in\_sources  
     (clean\_source), 9  
 set(opts), 50  
 show\_attribute\_table, 76  
 show\_inductive\_code\_tree, 77  
 show\_inductive\_code\_tree(), 43, 44  
 stats::heatmap(), 28  
 stripCodePathRoot, 78  
 strwrap(), 79  
 vecTxt, 78  
 vecTxtQ(vecTxt), 78  
 wrapVector, 79

`yaml_delimiter_indices`, 80