# Package ‘rock’

October 19, 2020

**Title** Reproducible Open Coding Kit

**Version** 0.1.1

**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

**LazyData** true

**RoxygenNote** 6.1.1

**Depends** R (>= 3.0.0)

**Imports** data.tree (>= 0.7.8), dplyr (>= 0.7.8), DiagrammeR (>= 1.0.0), glue (>= 1.3.0), graphics (>= 3.0.0), purrr (>= 0.2.5), stats (>= 3.0.0), utils (>= 3.5.0), yum (>= 0.0.1)

**Suggests** covr, knitr, rENA (>= 0.1.6), rmarkdown, testthat

**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** 2020-10-18 22:10:02 UTC

## R topics documented:

---

`add_html_tags` *Add HTML tags to a source*

---

### Description

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

### Usage

```
add_html_tags(x, codeClass = rock::opts$get(codeClass),
  idClass = rock::opts$get(idClass),
  sectionClass = rock::opts$get(sectionClass),
  uidClass = rock::opts$get(uidClass),
  utteranceClass = rock::opts$get(utteranceClass))
```

## Arguments

| | |
|---|---|
| `x` | A character vector with the source |
| `codeClass, idClass, sectionClass, uidClass, utteranceClass` | The classes to use for, respectively, codes, identifiers (such as case identifiers or coder identifiers), section breaks, utterance identifiers, 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.

---

`apply_graph_theme` *Apply multiple DiagrammeR global graph attributes*

---

## Description

Apply multiple DiagrammeR global graph attributes

## Usage

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

## Arguments

| | |
|---|---|
| `graph` | The DiagrammeR::DiagrammeR graph to apply the attributes to. |
| `...` | One or more character vectors of length three, where the first element is the attribute, the second the value, and the third, the attribute type (`graph`, `node`, or `edge`). |

## Value

The DiagrammeR::DiagrammeR graph.

## Examples

```
exampleSource <- '
---
codes:
  -
    id: parentCode
    label: Parent code
    children:
      -
        id: childCode1
      -
        id: childCode2
  -
```

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

## End(Not run)
```

---

`base30toNumeric` *Conversion between base10 and base30*

---

## Description

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

## Usage

```
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], "'.");
```

---

`clean_source` *Cleaning & editing sources*

---

## Description

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

## Usage

```
clean_source(input, output = NULL,
  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))

clean_sources(input, output, filenamePrefix = "", filenameSuffix = "",
  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", silent = FALSE)

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

## Arguments

| | |
|---|---|
| `input` | For `clean_source` and `search_and_replace_in_source`, either a character vector containing the text of the relevant source *or* a path to a file that contains the source text; for `clean_sources` and `search_and_replace_in_sources`, a path to a directory that contains the sources to clean. |
| `output` | For `clean_source` and `search_and_replace_in_source`, if not `NULL`, this is the name (and path) of the file in which to save the processed source (if it *is* `NULL`, the result will be returned visibly). For `clean_sources` and `search_and_replace_in_sources`, `output` is mandatory and is the path to the directory where to store the processed sources. This path will be created with a warning if it does not exist. An exception is if `"same"` is specified - in that case, every file will be written to the same directory it was read from. |
| `replacementsPre, replacementsPost` | Each is a list of two-element vectors, where the first element in each vector contains a regular expression to search for in the source(s), and the second element contains the replacement (these are passed as `perl` regular expressions; see `regex` for more information). Instead of regular expressions, simple words or phrases can also be entered of course (since those are valid regular expressions). `replacementsPre` are executed before the `utteranceSplits` are applied; `replacementsPost` afterwards. |
| `extraReplacementsPre, extraReplacementsPost` | To perform more replacements than the default set, these can be conveniently specified in `extraReplacementsPre` and `extraReplacementsPost`. This prevents you from having to manually copypaste the list of defaults to retain it. |
| `removeNewlines` | Whether to remove all newline characters from the source before starting to clean them. |
| `utteranceSplits` | This is a vector of regular expressions that specify where to insert breaks between utterances in the source(s). Such breakes are specified using `utteranceMarker`. |

| | |
|---|---|
| `preventOverwriting` | Whether to prevent overwriting of output files. |
| `encoding` | The encoding of the source(s). |
| `silent` | Whether to suppress the warning about not editing the cleaned source. |
| `filenamePrefix, filenameSuffix` | The prefix and suffix to add to the filenames when writing the processed files to disk. |
| `recursive` | Whether to search all subdirectories (`TRUE`) as well or not. |
| `filenameRegex` | A regular expression to match against located files; only files matching this regular expression are processed. |
| `replacements` | The strings to search & replace, as a list of two-element vectors, where the first element in each vector contains a regular expression to search for in the source(s), and the second element contains the replacement (these are passed as `perl` regular expressions; see `regex` for more information). Instead of regular expressions, simple words or phrases can also be entered of course (since those are valid regular expressions). |

## 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));

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"))));
```

## code_source    *Add one or more codes to one or more sources*

### Description

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

### Usage

```
code_source(input, codes, indices = NULL, codeDelimiters = c("[[",
  "]]"), silent = TRUE)

code_sources(input, codes, silent = FALSE)
```

### Arguments

| | |
|---|---|
| `input` | The source, or list of sources, as produced by one of the `loading_sources` functions. |
| `codes` | A named character vector, where each element is the code to be added to the matching utterance, and the corresponding name is either an utterance identifier (in which case the utterance with that identifier will be coded with that code), a code (in which case all utterances with that code will be coded with the new code as well), a digit (in which case the utterance at that line number in the source will be coded with that code), or a regular expression, in which case all utterances matching that regular expression will be coded with that source. If specifying an utterance ID or code, make sure that the code delimiters are included (normally, two square brackets). |

| | |
|---|---|
| `indices` | A logical vector of the same length as `input` that indicates to which utterance the code in `codes` should be applied. Note that if `indices` is provided, only the first element of `codes` is used, and its name is ignored. |
| `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. |
| `silent` | Whether to be chatty or quiet. |

## 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]);
```

---

collapse_occurrences *Collapse the occurrences in utterances into groups*

---

## Description

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

## Usage

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

## Arguments

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

## Value

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

## Examples

```
### 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 ann overview of all coded fragments per code.

### Usage

```
collect_coded_fragments(x, codes = ".*", context = 0, heading = NULL,
  headingLevel = 2, add_html_tags = TRUE, cleanUtterances = FALSE,
  output = NULL, template = "default", rawResult = FALSE,
  preventOverwriting = rock::opts$get(preventOverwriting),
  silent = rock::opts$get(silent))
```

### Arguments

| | |
|---|---|
| `x` | The parsed source(s) as provided by `rock::parse_source` or `rock::parse_sources`. |
| `codes` | The regular expression that matches the codes to include |
| `context` | How many utterances before and after the target utterances to include in the fragments. |
| `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. |
| `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. |
| `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(collect_coded_fragments(parsedExample));

### Only for the codes containing 'Code2'
cat(collect_coded_fragments(parsedExample,
                            'Code2'));
```

## `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 `rock::parse_source` or `rock::parse_sources`. |
| `codes` | The codes to include; by default, takes all codes. |
| `plotHeatmap` | Whether to plot the heatmap. |

### Value

The co-occurrence matrix; a `matrix`.

### Examples

```
### 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")
```

### Arguments

| | |
|---|---|
| `template` | The template to load; either the name of one of the ROCK templates (currently, only 'default' is available), or the path and filename of a CSS file. |

### Value

A character vector with the HTML fragment.

---

## `export_to_html` *Export parsed sources to HTML or Markdown*

---

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

## 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.

---

## `generate_uids` *Generate utterance identifiers (UIDs)*

---

### Description

This function generated utterance identifiers.

### Usage

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

### Arguments

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

### Value

A vector of UIDs.

### Examples

```
generate_uids(5);
```

---

## `load_source` *Load a source from a file or a string*

---

### 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 = "UTF-8", silent = FALSE)

load_sources(input, encoding = "UTF-8", filenameRegex = ".*",
  ignoreRegex = NULL, recursive = TRUE, full.names = FALSE,
  silent = FALSE)
```

### Arguments

| | |
|---|---|
| `input` | The filename or contents of the source for `load_source` and the directory containing the sources for `load_sources`. |
| `encoding` | The encoding of the file(s). |
| `silent` | Whether to be chatty or quiet. |
| `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. |
| `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`.

## `merge_sources` *Merge source files by different coders*

### Description

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

### Usage

```
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 imediately try to build the deductive tree(s) based on the information in this file (`FALSE`) or whether to skip that. Skipping this is useful if the full tree information is distributed over multiple files (in which case you should probably call `parse_sources` instead of `parse_source`). |
| `ignoreOddDelimiters` | If an odd number of YAML delimiters is encountered, whether this should result in an error (`FALSE`) or just be silently ignored (`TRUE`). |
| `preventOverwriting` | Whether to prevent overwriting existing files or not. |
| `encoding` | The encoding of the file to read (in `file`). |
| `silent` | Whether to provide (`FALSE`) or suppress (`TRUE`) more detailed progress updates. |
| `inheritSilence` | If not silent, whether to let functions called by `merge_sources` inherit that setting. |

## Value

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

## `opts` *Options for the rock package*

### Description

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

### Usage

```
opts
```

### Format

An object of class `list` of length 4.

### Details

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

The following arguments can be passed:

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

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

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

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.

**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.

**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).

**Two** Second item

## 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$convenience$metadataVars)
```

## Arguments

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

## Value

The result of a call to `rENA::ena.plot.network()`.

## 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);
```

---

parse_source *Parsing sources*

---

## Description

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

## Usage

```
parse_source(text, file, ignoreOddDelimiters = FALSE,
  postponeDeductiveTreeBuilding = FALSE,
  encoding = rock::opts$get(encoding), silent = rock::opts$get(silent))

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

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

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

## S3 method for class 'rockParsedSources'
plot(x, ...)
```

## Arguments

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

| | |
|---|---|
| | file; and if you're specifying a file and you want to be certain that an error is thrown if it doesn't exist, make sure to name it `file`. |
| `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 imediately 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`). |
| `encoding` | The encoding of the file to read (in `file`). |
| `silent` | Whether to provide (`FALSE`) or suppress (`TRUE`) more detailed progress updates. |
| `x` | The object to print. |
| `prefix` | The prefix to use before the 'headings' of the printed result. |
| `...` | Any additional arguments are passed on to the default print method. |
| `path` | The path containing the files to read. |
| `extension` | The extension of the files to read; files with other extensions will be ignored. Multiple extensions can be separated by a pipe (`\|`). |
| `regex` | Instead of specifing an extension, it's also possible to specify a regular expression; only files matching this regular expression are read. If specified, `regex` takes precedece over `extension`, |
| `recursive` | Whether to also process subdirectories (`TRUE`) or not (`FALSE`). |

## 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, encoding = "UTF-8",
  silent = TRUE)

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

## Arguments

| | |
|---|---|
| `input` | For `parse_source_by_coderId`, either a character vector containing the text of the relevant source *or* a path to a file that contains the source text; for `parse_sources_by_coderId`, a path to a directory that contains the sources to parse. |
| `ignoreOddDelimiters` | If an odd number of YAML delimiters is encountered, whether this should result in an error (`FALSE`) or just be silently ignored (`TRUE`). |
| `postponeDeductiveTreeBuilding` | Whether to imediately 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`). |
| `encoding` | The encoding of the file to read (in `file`). |
| `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 prepending 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(),
  preventOverwriting = rock::opts$get(preventOverwriting),
  encoding = rock::opts$get(encoding), silent = rock::opts$get(silent))

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

### Arguments

| | |
|---|---|
| `input` | The filename or contents of the source for `prepend_ids_to_source` and the directory containing the sources for `prepend_ids_to_sources`. |
| `output` | The filename where to write the resulting file for `prepend_ids_to_source` and the directory where to write the resulting files for `prepend_ids_to_sources` |
| `origin` | The time to use for the first identifier. |
| `preventOverwriting` | Whether to overwrite existing files (`FALSE`) or prevent that from happening (`TRUE`). |
| `encoding` | The encoding of the file(s). |
| `silent` | Whether to be chatty or quiet. |

### Value

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

## Examples

```
prepend_ids_to_source(input = "brief\nexample\nsource");
```

---

`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 Reprocucible 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.

---

`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 `firstElements`, and last `lastElements` elements. |

| | |
|---|---|
| `useQuote` | This character string is pre- and appended to all elements; so use this to quote all elements (`useQuote="'"`), doublequote all elements (`useQuote='"'`), or anything else (e.g. `useQuote='|'`). The only difference between `vecTxt` and `vecTxtQ` is that the latter by default quotes the elements. |
| `firstElements, lastElements` | The number of elements for which to use the first respective last delimiters |
| `lastHasPrecedence` | If the vector is very short, it's possible that the sum of firstElements and lastElements is larger than the vector length. In that case, downwardly adjust the number of elements to separate with the first delimiter (`TRUE`) or the number of elements to separate with the last delimiter (`FALSE`)? |
| `...` | Any addition arguments to `vecTxtQ` are passed on to `vecTxt`. |

## Value

A character vector of length 1.

## Examples

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

# Index