

# Package ‘rock’

June 14, 2019

**Title** Reproducible Open Coding Kit

**Version** 0.0.1

**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 a decentralized construct taxonomy ('DCT').

**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]

**Repository** CRAN

**Date/Publication** 2019-06-14 11:30:03 UTC

**R topics documented:**

apply_graph_theme . . . . .	2
base30toNumeric . . . . .	3
cat0 . . . . .	4
clean_source . . . . .	5
code_source . . . . .	7
collapse_occurrences . . . . .	8
collect_coded_fragments . . . . .	10
create_cooccurrence_matrix . . . . .	11
generate_uids . . . . .	12
load_source . . . . .	13
parsed_sources_to_ena_network . . . . .	13
parse_source . . . . .	14
prepend_ids_to_source . . . . .	17
repeatStr . . . . .	18
rock . . . . .	19
vecTxt . . . . .	20

<b>Index</b>	<b>21</b>
--------------	-----------

---

apply_graph_theme	<i>Apply multiple DiagrammeR global graph attributes</i>
-------------------	--

---

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

---

clean_source	<i>Cleaning sources</i>
--------------	-------------------------

---

## Description

This function can be used to 'clean' one or more sources. 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 = list(c("([^\s.])\s+\.")("^\s.]),
    "\1.\3"), c("([^\s.])\s+\.")("^\s.]),
    "\1.\3"), c("\s*\r?\n){3,}", "\n")),
  extraReplacementsPre = NULL,
  utteranceSplits = c("([\?!\!]+\s?|. . . \s?|[[:alnum:]]\s?)\s+(?!.\s.)\s?"),
  utteranceMarker = "\n",
  replacementsPost = list(c("([^\s,])", "^\s]"), "\1, \2")),
  extraReplacementsPost = NULL, preventOverwriting = TRUE,
  removeNewlines = FALSE, encoding = "UTF-8", silent = FALSE)
```

```
clean_sources(input, output,
  replacementsPre = list(c("([^\s.])\s+\.")("^\s.]),
    "\1.\3"), c("([^\s.])\s+\.")("^\s.]),
    "\1.\3"), c("\s*\r?\n){3,}", "\n")),
  extraReplacementsPre = NULL,
  utteranceSplits = c("([\?!\!]+\s?|. . . \s?|[[:alnum:]]\s?)\s+(?!.\s.)\s?"),
  utteranceMarker = "\n",
  replacementsPost = list(c("([^\s,])", "^\s]"), "\1, \2")),
  extraReplacementsPost = NULL, preventOverwriting = TRUE,
  removeNewlines = FALSE, encoding = "UTF-8", silent = FALSE)
```

## Arguments

input	For <code>clean_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> , a path to a directory that contains the sources to clean.
output	For <code>clean_source</code> , if not <code>NULL</code> , this is the name (and path) of the file in which to save the cleaned source (if it <i>is</i> <code>NULL</code> , the result will be returned visible). For <code>clean_sources</code> , <code>output</code> is mandatory and is the path to the directory where to store the cleaned sources. This path will be created with a warning if it does not exist.
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.

`utteranceSplits`

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

`utteranceMarker`

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

`preventOverwriting`

Whether to prevent overwriting of output files.

`removeNewlines` Whether to remove all newline characters from the source before starting to clean them.

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

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

## Details

When called with its default arguments, the following will happen:

- 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));
```

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**

<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).
<code>indices</code>	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>indiced</code> is provided, only the first element of <code>codes</code> is used, and its name is ignored.
<code>codeDelimiters</code>	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.
<code>silent</code>	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\\d\\s" = "timeRef");

### Apply rules
codedExample <- code_source(loadedExample,
                             codeSpecs);

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

### Also add code "foo" to utteranced 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]);

```



**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, heading = NULL,
  headingLevel = 2, rawResult = FALSE, output = NULL,
  cleanUtterances = TRUE, silent = TRUE)
```

### 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.
heading	Optionally, a title to include in the output. The title will be prefixed with <code>headingLevel</code> hashes (#), and the codes with <code>headingLevel+1</code> hashes. If <code>NULL</code> (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 <code>NULL</code> or a character vector (e.g. <code>FALSE</code> ). If no heading is used, the code prefix will be <code>headingLevel</code> hashes, instead of <code>headingLevel+1</code> hashes.
headingLevel	The number of hashes to insert before the headings.
rawResult	Whether to return the raw result, a list of the fragments, or one character value in markdown format.
output	Here, a path and filename can be provided where the result will be written. If provided, the result will be returned invisibly.
cleanUtterances	Whether to use the clean or the raw utterances when constructing the fragments (the raw versions contain all codes).
silent	Whether to provide ( <code>FALSE</code> ) or suppress ( <code>TRUE</code> ) more detailed progress updates.

### 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 <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);
```

---

generate\_uids

*Generate utterance identifiers (UIDs)*

---

## Description

Generate utterance identifiers (UIDs)

## 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 <code>numericToBase30()</code> . By default, the present time is used as origin, one one centisecond is added for every identifier to generate. <code>origin</code> 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	<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 = "UTF-8", silent = FALSE)
```

```
load_sources(input, encoding = "UTF-8", silent = FALSE)
```

### 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.

### Value

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

---

parsed_sources_to_ena_network	<i>Create an ENA network out of one or more parsed sources</i>
-------------------------------	--

---

### 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 <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
parsedExamples$mergedSourceDf$units <- rep(1:11, each=9);

### 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, codeRegexes = c(codes =
  "\\[\\[([a-zA-Z0-9._>-]+)\\]\\]", idRegexes = c(caseId =
  "\\[\\[cid=([a-zA-Z0-9._-]+)\\]\\]", stanzaId =
  "\\[\\[sid=([a-zA-Z0-9._-]+)\\]\\]"),
  sectionRegexes = c(paragraphs = "---paragraph-break---", secondary =
  "---<[a-zA-Z0-9]?>---"), autoGenerateIds = c("stanzaId"),
  persistentIds = c("caseId"), noCodes = "^uid:|^dct:|^ci:",
  inductiveCodingHierarchyMarker = ">",
  metadataContainers = c("metadata"), codesContainers = c("codes",
  "dct"), delimiterRegex = "^---$", ignoreRegex = "^#",
  ignoreOddDelimiters = FALSE, encoding = "UTF-8",
  postponeDeductiveTreeBuilding = FALSE, silent = TRUE)

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

parse_sources(path, extension = "rock|dct", regex = NULL,
  recursive = TRUE, codeRegexes = c(codes =
  "\\[\\[([a-zA-Z0-9._>-]+)\\]\\]", idRegexes = c(caseId =
  "\\[\\[cid=([a-zA-Z0-9._-]+)\\]\\]", stanzaId =
  "\\[\\[sid=([a-zA-Z0-9._-]+)\\]\\]"),
  sectionRegexes = c(paragraphs = "---paragraph-break---", secondary =
  "---<[a-zA-Z0-9]?>---"), autoGenerateIds = c("stanzaId"),
  persistentIds = c("caseId"), noCodes = "^uid:|^dct:|^ci:",
  inductiveCodingHierarchyMarker = ">",
  metadataContainers = c("metadata"), codesContainers = c("codes",
  "dct"), delimiterRegex = "^---$", ignoreRegex = "^#",
  ignoreOddDelimiters = FALSE, encoding = "UTF-8", silent = TRUE)

## 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.

codeRegexes, idRegexes, sectionRegexes	These are named character vectors with one or more regular expressions. For codeRegexes, these specify how to extract the codes (that were used to code the sources). For idRegexes, these specify how to extract the different types of identifiers. For sectionRegexes, these specify how to extract the different types of sections. The codeRegexes and idRegexes must each contain one capturing group to capture the codes and identifiers, respectively.
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 <b>ignored</b> , 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.
metadataContainers	The name of YAML fragments containing metadata (i.e. attributes about cases).
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.
ignoreOddDelimiters	If an odd number of YAML delimiters is encountered, whether this should result in an error (FALSE) or just be silently ignored (TRUE).
encoding	The encoding of the file to read (in file).
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).
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, regex takes precedence 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;

```

---

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

prepend_ids_to_sources(input, output = NULL, preventOverwriting = TRUE,
  encoding = "UTF-8", origin = Sys.time(), silent = FALSE)
```

**Arguments**

input	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> .
output	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>
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 is specified) or visibly (if not).

**Examples**

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

---

repeatStr	<i>Repeat a string a number of times</i>
-----------	--

---

**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.

---

 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

`apply_graph_theme`, 2

`base30conversion` (`base30toNumeric`), 3

`base30toNumeric`, 3

`base::readLines()`, 15

`base::strsplit()`, 15

`cat`, 4

`cat0`, 4

`clean_source`, 5

`clean_source()`, 19

`clean_sources` (`clean_source`), 5

`clean_sources()`, 13, 17, 19

`code_source`, 7

`code_sources` (`code_source`), 7

`collapse_occurrences`, 8

`collapse_occurrences()`, 19

`collect_coded_fragments`, 10

`collect_coded_fragments()`, 19

`create_cooccurrence_matrix`, 11

`create_cooccurrence_matrix()`, 19

`DiagrammeR::DiagrammeR`, 2

`generate_uids`, 12

`generate_uids()`, 3

`load_source`, 13

`load_sources` (`load_source`), 13

`numericToBase30` (`base30toNumeric`), 3

`numericToBase30()`, 12

`parse_source`, 14

`parse_source()`, 9, 19

`parse_sources` (`parse_source`), 14

`parse_sources()`, 19

`parsed_sources_to_ena_network`, 13

`plot.rockParsedSources` (`parse_source`), 14

`prepend_ids_to_source`, 17

`prepend_ids_to_source()`, 19

`prepend_ids_to_sources` (`prepend_ids_to_source`), 17

`prepend_ids_to_sources()`, 19

`prepending_uids` (`prepend_ids_to_source`), 17

`prepending_uids()`, 13

`print.rockParsedSource` (`parse_source`), 14

`print.rockParsedSources` (`parse_source`), 14

`regex`, 6

`rENA::ena.plot.network()`, 14

`repeatStr`, 18

`repStr` (`repeatStr`), 18

`rock`, 19

`rock-package` (`rock`), 19

`stats::heatmap()`, 11

`vecTxt`, 20

`vecTxtQ` (`vecTxt`), 20