Package ‘rapport’

November 18, 2015

Maintainer Gergely Daróczi <daroczig@rapporter.net>

Title A Report Templating System

Type Package

Encoding UTF-8

Description Facilitating the creation of reproducible statistical report templates. Once created, rapport templates can be exported to various external formats (HTML, LaTeX, PDF, ODT etc.) with pandoc as the converter backend.

Author Aleksandar Blagotić <alex@rapporter.net> and Gergely Daróczi <daroczig@rapporter.net>

Version 1.0

Date 2015-11-15

URL http://rapport-package.info/

BugReports https://github.com/rapporter/rapport/issues

License AGPL-3

LazyData yes

LazyLoad yes

Depends R (>= 2.15.0)

Imports grDevices, utils, yaml, stringr, rapportools, pander

SystemRequirements pandoc (http://johnmacfarlane.net/pandoc) for exporting markdown files to other formats.

NeedsCompilation no

Repository CRAN

Date/Publication 2015-11-18 09:38:09
### R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>rapport-package</td>
<td>3</td>
</tr>
<tr>
<td>as.character.rapport.inputs</td>
<td>4</td>
</tr>
<tr>
<td>as.character.rapport.meta</td>
<td>4</td>
</tr>
<tr>
<td>as.yaml.bool</td>
<td>5</td>
</tr>
<tr>
<td>check.input.value</td>
<td>5</td>
</tr>
<tr>
<td>check.input.value.class</td>
<td>6</td>
</tr>
<tr>
<td>check.report.chunks</td>
<td>6</td>
</tr>
<tr>
<td>check.tpl</td>
<td>7</td>
</tr>
<tr>
<td>extract.meta</td>
<td>7</td>
</tr>
<tr>
<td>get.tags</td>
<td>8</td>
</tr>
<tr>
<td>guess.input</td>
<td>9</td>
</tr>
<tr>
<td>guess.input.description</td>
<td>9</td>
</tr>
<tr>
<td>guess.input.label</td>
<td>10</td>
</tr>
<tr>
<td>guess.input.name</td>
<td>10</td>
</tr>
<tr>
<td>guess.l</td>
<td>10</td>
</tr>
<tr>
<td>guess.old.input.length</td>
<td>11</td>
</tr>
<tr>
<td>guess.old.input.type</td>
<td>11</td>
</tr>
<tr>
<td>inputs-deprecated</td>
<td>12</td>
</tr>
<tr>
<td>is.rapport</td>
<td>14</td>
</tr>
<tr>
<td>ius2008</td>
<td>14</td>
</tr>
<tr>
<td>print.rapport</td>
<td>16</td>
</tr>
<tr>
<td>print.rapport.info</td>
<td>16</td>
</tr>
<tr>
<td>print.rapport.inputs</td>
<td>17</td>
</tr>
<tr>
<td>print.rapport.meta</td>
<td>17</td>
</tr>
<tr>
<td>rapport</td>
<td>18</td>
</tr>
<tr>
<td>rapport-helpers</td>
<td>19</td>
</tr>
<tr>
<td>rapport.body</td>
<td>21</td>
</tr>
<tr>
<td>rapport.check.template</td>
<td>22</td>
</tr>
<tr>
<td>rapport.docx</td>
<td>22</td>
</tr>
<tr>
<td>rapport.example</td>
<td>23</td>
</tr>
<tr>
<td>rapport.export</td>
<td>23</td>
</tr>
<tr>
<td>rapport.header</td>
<td>25</td>
</tr>
<tr>
<td>rapport.html</td>
<td>26</td>
</tr>
<tr>
<td>rapport.info</td>
<td>26</td>
</tr>
<tr>
<td>rapport.inputs</td>
<td>27</td>
</tr>
<tr>
<td>rapport.ls</td>
<td>29</td>
</tr>
<tr>
<td>rapport.meta</td>
<td>30</td>
</tr>
<tr>
<td>rapport.odt</td>
<td>31</td>
</tr>
<tr>
<td>rapport.path</td>
<td>31</td>
</tr>
<tr>
<td>rapport.path.add</td>
<td>32</td>
</tr>
<tr>
<td>rapport.path.remove</td>
<td>32</td>
</tr>
<tr>
<td>rapport.path.reset</td>
<td>33</td>
</tr>
<tr>
<td>rapport.pdf</td>
<td>33</td>
</tr>
<tr>
<td>rapport.read</td>
<td>34</td>
</tr>
<tr>
<td>rapport.renew</td>
<td>34</td>
</tr>
<tr>
<td>rapport.rerun</td>
<td>35</td>
</tr>
</tbody>
</table>
Description

`rapport` is an R package that facilitates creation of reproducible statistical report templates. Once created, `rapport` templates can be exported to various external formats: HTML, LaTeX, PDF, ODT, etc.

Details

For detailed introductions please check out our homepage: [http://rapport-package.info](http://rapport-package.info).

You may use the package-bundled templates with a minimal R knowledge - a quick tutorial is shown in the package demo: `demo(rapport, ask = FALSE)`.

Apart from R, all you need to know to start writing your own templates is Pandoc’s markup syntax, and several `rapport`-specific conventions that allow the reproducible of the template. `rapport` uses `brew`-like tags to support dynamic inline and/or block evaluation of R code. Unlike many other report-writing conventions in R (`Sweave`, `brew`, `knitr`), `rapport` converts generated output to a convenient form via `pander` package and pandoc as the document converting backend. `rapport` also comes with support for plots: images are automatically saved to temporary file, and image path is returned or redrawn on demand.

The output of `rapport` command depends on various package-specific options. Please skim through the summary of following options:

- **rapport.user**: a (user)name to show in exported report (defaults to "Anonymous")
- **rapport.file.name**: a general filename of generated images and exported documents without extension. Some helper pseudo-code would be replaced with handy strings while running `rapport` and `rapport.export`:
  - `%t`: unique random character strings based on `tempfile`,
  - `%T`: template name in action,
  - `%n`: an auto-increment integer based on similar (plot) file names (see: `?evalsOptions`),
  - `%N`: an auto-increment integer based on similar exported document’s file name
- **rapport.file.path**: a directory where generated images and exported documents would take place.
- By default `rapport` function saves plots to image files (see the settings in `evalsOptions()`) and print method just shows the path(s) of the generated image(s). If you would like to see the plot(s) when calling `rapport` function from an interactive R console, please set `evalsOptions('graph.recordplot')` and the global `rapport.graph.replay` option to TRUE beforehand. In that case all generated plots will be displayed after printing the `rapport` object. These options are set to FALSE by default although we find these settings really handy, as you can resize the images on the fly and export resized images to HTML/ODT/DOCX/PDF etc. If you would even like to save the actual environment of each generated plot (variables, data sets etc.) as an RData file, please set `evalsOptions('graph.env')` to TRUE.
• `rapport` also has some options to set formatting style of numbers, characters and dates specified in `panderOptions()
• the exported graphs can be customised via further `panderOptions`

See Also


---

description

Converts template inputs to character vector with YAML strings.

Usage

```r
## S3 method for class 'rapport.inputs'
as.character(x, ...)
```

Arguments

- `x` template inputs object
- `...` ignored

---

description

Converts template metadata to character vector with YAML strings.

Usage

```r
## S3 method for class 'rapport.meta'
as.character(x, ...)
```

Arguments

- `x` template metadata object
- `...` ignored
as.yaml.bool

Convert YAML booleans to R ones

Description

We need this because of the silly R/YAML bug. Chillax, it's for internal use only, and since we're about to call it on bunch of places, we needed a function.

Usage

as.yaml.bool(x)

Arguments

x a character vector with YAML booleans

check.input.value

Check input value

Description

A bit misleading title/function name - it validates input values, according to rules set in general input attributes (length) or class-specific ones (nchar, nlevels or limit).

Usage

check.input.value(input, value = NULL, attribute.name = c("length", "nchar", "nlevels", "limit"))

Arguments

input input item
value input value, either template-defined, or set by the user
attribute.name input attributes containing validation rules (defaults to length)
check.input.value.class

*Check Input Value Class*

**Description**
Checks the class of an input value.

**Usage**

```r
ccheck.input.value.class(value, class = c("character", "complex", "factor", "integer", "logical", "numeric", "raw"), input.name = NULL)
```

**Arguments**

- `value` input value
- `class` input class (defaults to `NULL`)
- `input.name` input name (used in messages)

check.report.chunks

*Check Report Chunks*

**Description**
Checks for warnings and errors in report chunks.

**Usage**

```r
ccheck.report.chunks(rp, what = c("errors", "warnings", "messages"))
```

**Arguments**

- `rp` rapport object
- `what` what fields to check. defaults to all
check.tpl

---

**Check template validity**

**Description**

Throw error

**Usage**

```r
check.tpl(txt, open.tag = get.tags("header.open"),
            close.tag = get.tags("header.close"), ...)
```

**Arguments**

- `txt`: character vector with template contents
- `open.tag`: opening tag regexp
- `close.tag`: closing tag regexp
- `...`: additional params for tag matching (see `grep`)

---

**Extract Template Metadata**

**Description**

Check if template metadata field matches provided format, and return matched value in a list.

**Usage**

```r
extract.meta(x, title, regex, short = NULL, trim.white = TRUE,
             mandatory = TRUE, default.value = NULL, field.length = 1000, ...)
```

**Arguments**

- `x`: a string containing template metadata
- `title`: a string containing metadata field title (can be regex-powered)
- `regex`: a string with regular expression to match field value
- `short`: a string with a short name for given metadata field
- `trim.white`: a logical value indicating whether trailing and leading spaces of the given string should be removed before extraction
- `mandatory`: a logical value indicating required field
- `default.value`: fallback to this value if non-mandatory field is not found/malformed
- `field.length`: maximum number of field characters (defaults to 1000)
- `...`: additional parameters for `grep` function
**get.tags**

**Value**

a list with matched content, or null if the field is not required

**Examples**

```r
## Not run:
rapport:::extract.meta("Name: John Smith", "Name", "[:alpha:]+([[:alpha:]]+)?")
## $name
## [1] "John Smith"

rapport:::extract.meta("Name: John", "Name", "[:alpha:]+([[:alpha:]]+)?")
## $name
## [1] "John"

## End(Not run)
```

---

### Tag Values

**Description**

Returns report tag values (usually regexes): either user-defined, or the default ones.

**Usage**

```r
get.tags(tag.type = c("all", "header.open", "header.close", "comment.open", "comment.close"), preset = c("user", "default"))
```

**Arguments**

- `tag.type` a character value with tag value name
- `preset` a character value specifying which preset to return

**Details**

Default parameters are read from options:

- 'header.open'
- 'header.close'
- 'comment.open'
- 'comment.close'

**Value**

either a list (default) or a character value with tag regexes
Examples

```r
## Not run:
get.tags()       # same as 'get.tags("all")'
get.tags("header.open")

## End(Not run)
```

---

**guess.input**  
*Guess Input*

---

**Description**

Checks and returns valid input from YAML input definition.

**Usage**

```r
guess.input(input)
```

**Arguments**

- `input`  
a named list containing input definition

---

**guess.input.description**  
*Input Description*

---

**Description**

Checks and returns input description.

**Usage**

```r
guess.input.description(description)
```

**Arguments**

- `description`  
a character string containing input description
guess.input.label  **Input Label**

**Description**
Checks and returns input label.

**Usage**
guess.input.label(label)

**Arguments**

- label  a character string containing input label

---

guess.input.name  **Input Name Validation**

**Description**
From v.0.51 one or more characters that are not newline should do the trick. Note that white spaces will be trimmed from both ends in resulting string.

**Usage**
guess.input.name(name)

**Arguments**

- name  a character value with input name

---

guess.l  **Guess length-like fields**

**Description**
Since length, nchar, nlevels and limit have (almost) same format,

**Usage**
guess.l(len = c("length", "nchar", "nlevels", "limit"),
input.name = NULL, limit.class = c("numeric", "integer"))
Arguments

- **len**: length field value, either a number or a named list
- **type**: type of length-like field
- **input.name**: input name
- **limit.class**: input class to perform limit-specific checks

---

**guess.old.input.length**

*Deprecated input limits*

---

**Description**

Guess deprecated input length.

**Usage**

```r
guess.old.input.length(x, input.type)
```

**Arguments**

- **x**: a character string containing input length definition
- **input.type**: a character string containing input type

---

**guess.old.input.type**

*Check Type*

---

**Description**

Checks type of template input, based on provided string. If input definition is syntactically correct, a list is returned, containing input type, size limits, and default value (for CSV options and boolean types only).

**Usage**

```r
guess.old.input.type(x)
```

**Arguments**

- **x**: a character string containing input definition
Description

As of version 0.5, rapport relies on YAML syntax to define inputs. The following sections describe deprecated input definition syntax.

Details

Input Specifications

Apart from template metadata, header also requires specification for template inputs. In most cases, inputs refer to variable names in provided dataset, but some inputs have special meaning inside rapport, and some of them don’t have anything to do with provided dataset whatsoever. Most inputs can contain limit specification, and some inputs can also have a default value. At first we’ll explain input specifications on the fly, and in following sections we’ll discuss each part in thorough details. Let’s start with a single dummy input specification:

*foo.bar | numeric[1,6] | Numeric variable | A set of up to 6 numeric variables

Required Inputs

Asterisk sign (*) in front of an input name indicates a mandatory input. So it is possible to omit input (unless it’s required, of course), but you may want to use this feature carefully, as you may end up with ugly output. If an input isn’t mandatory, NULL is assigned to provided input name, and the object is stored in transient evaluation environment.

Input Name

rapport has its own naming conventions which are compatible, but different from traditional R naming conventions. Input name ("foo.bar" in previous example) must start with an alphabet letter, followed either by other alphabet letters or numbers, separated with _ or . For example, valid names are: foo.bar, f00_bar, or Fo0_bar.input. Input name length is limited on 30 characters by default. At any time you can check your desired input name with check.name function. Note that input names are case-sensitive, just like symbols in R.

Input Type

Input type is specified in the second input block. It is the most (read: "only") complex field in an input specification. It consists of type specification, limit specification and sometimes a default value specification. Most input types are compatible with eponymous R modes: character, complex, logical, numeric, or R classes like factor. Some are used as "wildcards", like variable, and some do not refer to dataset variables at all: boolean, number, string and option. Here we’ll discuss each input type thoroughly. We will use term variable to denote a vector taken from a dataset (for more details see documentation for is.variable). All inputs can be divided into two groups, depending on whether they require a dataset or not:

- dataset inputs:
  - character - matches a character variable
  - complex - matches a character variable
  - numeric - matches a numeric variable
– **factor** - matches a factor variable (i.e. R object of `factor` class)
– **variable** - matches any variable of previously defined types

**standalone inputs:**
– **string** - accepts an atomic character vector
– **number** - accepts an atomic numeric vector
– **boolean** - accepts a logical value
– **option** - accepts a comma-separated list of values, that are to be matched with `match.arg`.
The first value in a list is the default one.

Now we’ll make a little digression and talk about **input limits**. You may have noticed some additional stuff in type specification, e.g. `numeric[1,6]`. All dataset inputs, as well as *string* and *numeric standalone inputs* can contain **limit specifications**. If you want to bundle several variables from dataset or provide a vector with several string/numeric values, you can apply some rules within square brackets in `[a,b]` format, where `[a,b]` stands for "from a to b inputs", e.g. `[1,6]` means "from 1 to 6 inputs". Limit specifications can be left out, but even in that case implicit limit rules are applied - for variables, as well as boolean and option inputs it’s `[1,1]`, for strings `[1,256]` and for number inputs `[-Inf,Inf]`.

**Dataset inputs** will match one or more variables from a dataset, and check its mode and/or class. **variable** type is a bit different, since it matches any kind of variable (not to confuse with Any type), but it still refers to variable(s) from a provided dataset. Dataset inputs cannot have default value, but can be optional (just leave out * sign in front of input name). Note that if you provide more than one variable name in `rapport` function call, that input will be stored as a `data.frame`, otherwise, it will be stored as a `variable` (atomic vector).

**Standalone inputs** are a bit different since they do not refer to any variables from a dataset. However, they are more complex than *dataset inputs*, especially because they can contain default values.

– **number** and **string** inputs are defined with `number` and `string` respectively. They can also contain limit specifications, but the limits are treated in a slightly different manner. `number[-2.58,3]` will match any number within an interval from -2.58 to 3. If the limit specification is omitted, an implicit one are assigned `[-Inf,Inf]`. Limit specifications for string inputs define the range of characters that provided string can have, e.g. `string[1,6]` matches the string with at least 1 and at most 6 characters. If omitted, limit specifications for strings are implicitly set to `[1,256]`. `number` and `string` inputs can have **default value**, which can be defined by placing = after type/limit specification followed by default value. For instance, `number[1,6]=3.14` sets value 3.14 as default. Note that for number inputs an additional check will be applied to ensure that provided default number belongs to an interval defined in the limit specification `([1,6]=7 will throw an error)`. For string inputs, the default value `string=foo` sets "foo" as default string value (note that you don’t have to specify quotes unless they are the part of the default string). Default value will be checked to ensure that its length falls within the interval provided in the limit specification.

– **boolean** inputs can contain either TRUE or FALSE values. The specified value is the default one. They cannot contain limit specification, but implicitly the limits are set to `[1,1]`.

– **option** inputs are nothing more than a comma-separated list of strings. Even if you specify numbers in a list, they will be coerced to strings once the list is parsed. Values in **option** list will be placed in a character vector, and matched with `match.arg` function. That means that you could only choose one value from a list. Partial matches are allowed, and the first value in **option** list is the default one. Just like in **boolean** inputs, limits are implicitly set to `[1,1]`. 
**Input Label and Description**
Third block in input definition is an input label. While *variable* can have its own label (see *label*), you may want to use the one defined in input specifications. At last, fourth block contains input description, which should be a lengthy description of current input. Note that all the fields in input specification are mandatory. You can cheat, though, by providing a non-space character (e.g. a dot) as an input label and/or description, but please don’t do that unless you’re testing the template. Labels and descriptions are meant to be informative.

---

**is.rapport**  
*Rapport Object*

**Description**
Checks if provided R object is of `rapport` class.

**Usage**

```r
is.rapport(x)
```

**Arguments**

- `x` any R object to check

**Value**

a logical value indicating whether provided object is a `rapport` object

---

**ius2008**  
*Internet Usage Survey*

**Description**
This dataset contains data gathered in a survey of Internet usage in Serbian population in the period from April to May 2008. During 90-day period, there were gathered 709 valid responses via on-line distributed questionnaire.

**Details**
However, this dataset does not contain the original data, as some random noise is added afterwards, in order to demonstrate functionality of `rapport` helpers.

Dataset variables can be divided into 3 sets: demographic data, Internet usage aspects and application usage/content preference.

**Demographic variables**

- *gender* - respondent’s gender (factor with 2 levels: "male" and "female")
• age - respondent’s age
• dwell - dwelling (factor with 3 levels: "village", "small town" and "city")
• student - is respondent a student? (factor with 2 levels: "no" and "yes")
• partner - partnership status (factor with 3 levels: "single", "in a relationship" and "married")

Internet usage aspects
Following variables depict various aspects of Internet usage:

• edu - time spent on-line in educational purposes (expressed in hours)
• leisure - time spent on-line in leisure time (expressed in hours)
• net.required - is Internet access required for your profession? (factor with 5 levels: "never", "rarely", "sometimes", "often" and "always")
• net.pay - who pays for Internet access? (factor with 5 levels: "parents", "school/faculty", "employer", "self-funded" and "other")
• net.use - how long is respondent using Internet? (ordered factor with 7 levels, ranging from "less than 6 months" to "more than 5 years")

Application usage and on-line content preference
These variables include data on the use of Internet applications and content available on the Internet. Practically, they contain responses from a set of 8 questions on a five-point Likert scale.

• chatim - usage of chat and/or instant messaging applications
• game - usage of on-line games
• surf - frequency of web-surfing
• email - usage of e-mail applications
• download - frequency of file downloading
• forum - attendance at web-forums
• socnet - usage of social networking services
• xxx - traffic to pornographic websites

Author(s)
Aleksandar Blagotic <aca.blagotic@gmail.com>
Dusan Vuckovic <sylphs21125@gmail.com>

Examples

## Not run:
rapport("example", ius2008, var = "it.leisure")

## End(Not run)
print.rapport  

Prints rapport

Description

Default print method for rapport class objects that shows evaluated report contents.

Usage

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

Arguments

- `x` any "rapport" class object
- `...` ignored

Examples

```r
# Not run:
rappor('example', data = mtcars, var='hp')

# End(Not run)
```

print.rapport.info  

Print Template Header

Description

Prints out the contents of template header (both metadata and inputs) in human-readable format, so you can get insight about the template requirements.

Usage

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

Arguments

- `x` object of class rp.header. See rapport.header for details.
- `...` ignored
**print.rapport.inputs**  

*Print Template Inputs*

---

**Description**

Prints out the contents of template inputs in human-readable format.

**Usage**

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

**Arguments**

- `x` object of class `rapport.inputs`. See `rapport.inputs` for details.
- `...` ignored

---

**print.rapport.meta**  

*Print Template Metadata*

---

**Description**

Prints out the contents of template metadata in human-readable format.

**Usage**

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

**Arguments**

- `x` object of class `rapport.meta`. See `rapport.meta` for details.
- `...` ignored
rappor

Evaluate Template

Description

This is the central function in the rapport package, and hence eponymous. In following lines we’ll use rapport to denote the function, not the package. rapport requires a template file, while
dataset (data argument) can be optional, depending on the value of Data required field in template
header. Template inputs are matched with ... argument, and should be provided in x = value
format, where x matches input name and value, wait for it... input value! See rapport.inputs for
more details on template inputs.

Usage

rappor(fp, data = NULL, ..., env = .GlobalEnv, reproducible = FALSE,
header.levels.offset = 0, graph.output = evalsOptions("graph.output"),
file.name =getOption("rapport.file.name"),
file.path =getOption("rapport.file.path"),
graph.width = evalsOptions("width"),
graph.height = evalsOptions("height"), graph.res = evalsOptions("res"),
graph.hi.res = evalsOptions("hi.res"),
graph.replay = evalsOptions("rapport.graph.recordplot"))

Arguments

fp a template file pointer (see rapport:::rapport.read for details)
data a data.frame to be used in template
... matches template inputs in format 'key = "value"'
env the parent environment to be forked, in which temporary new.env template com-
mands be evaluated
reproducible a logical value indicating if the call and data should be stored in template object,
thus making it reproducible (see rapport.rerun for details)
header.levels.offset number added to header levels (handy when using nested templates)
graph.output the required file format of saved plots (optional)
file.name set the file name of saved plots and exported documents. A simple character
string might be provided where %N would be replaced by an auto-increment integer based on similar exported document’s file name , %n an auto-increment integer based on similar (plot) file names (see: ?evalsOptions), %T by the name of
the template in action and %t by some unique random characters based on
tempfile.
file.path path of a directory where to store generated images and exported reports
graph.width the required width of saved plots (optional)
graph.height the required height of saved plots (optional)
graph.res  the required nominal resolution in ppi of saved plots (optional)
graph.hi.res logical value indicating if high resolution (1280x~1280) images would be also generated
graph.replay logical value indicating if plots need to be recorded for later replay (eg. while printing rapport objects in R console)

Details
Default parameters are read from evalsOptions() and the following options:

- 'rapport.file.name',
- 'rapport.file.path',

Value
a list with rapport class.

See Also

rapport-package

Examples

## Not run:
rapport('Example', ius2008, v = "leisure")
rappor('Descriptives', ius2008, var = "leisure")

## generating high resolution images also
rapport('Example', ius2008, v = "leisure", graph.hi.res = TRUE)
rappor.html('NormalityTest', ius2008, var = "leisure", graph.hi.res=T)
## generating only high resolution image
rapport('Example', ius2008, v = "leisure", graph.width = 1280, graph.height = 1280)
## nested templates cannot get custom setting, use custom rapport option:
options('graph.hi.res' = TRUE)
rappor('AnalyzeWizard', data=ius2008, variables=c('edu', 'game'))

## End(Not run)
Details

Export helpers

- rapport.docx
- rapport.html
- rapport.odt
- rapport.pdf
- rapport.export

Please load the rapportools package if you would use any of the below functions in the .GlobalEnv, or simply add rapportools to the required packages section in your template file. That latter is a lot cleaner solution.

General purpose helpers

- adj.rle
- alike.integer
- capitalise
- catn
- fml
- is.boolean
- is.empty
- is.number
- is.string
- is.tabular
- is.variable
- messagef
- pct
- stopf
- tocamel
- trim.space
- vgsub
- warningf

Summary statistics

- rp.desc
- rp.freq

Univariate descriptive statistics

- rp.iqr
- rp.max
**rapport.body**

<table>
<thead>
<tr>
<th>Description</th>
<th>Template Body</th>
</tr>
</thead>
</table>

Returns contents of the template body.

**Usage**

```
rapport.body(fp, htag = get.tags("header.close"), ...)  
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp</td>
<td>a template file pointer (see rapport::rapport.read for details)</td>
</tr>
<tr>
<td>htag</td>
<td>a string with closing body tag</td>
</tr>
<tr>
<td>...</td>
<td>additional arguments to be passed to grep function</td>
</tr>
</tbody>
</table>

**Value**

a character vector with template body contents
rapport.check.template

*Check Rapport Template*

**Description**
Checks if the examples of given template can be run without any error.

**Usage**

\[\text{rapport.check.template}(\text{fp})\]

**Arguments**

- **fp**
  
  a character vector containing template name (".rapport" extension is optional), file path or a text to be split by line breaks

**Details**

If everything went fine and you get a list of success equals to TRUE values, otherwise success returns FALSE with additional message

**Examples**

```r
## Not run:
rapport.check.template("Example")

## End(Not run)
```

---

rapport.docx

*Rapport to DOCX*

**Description**

This is a simple wrapper around `rapport` and `rapport.export`. Basically it works like `rapport` but the returned class is exported at one go.

**Usage**

\[\text{rapport.docx}(\ldots)\]

**Arguments**

- \ldots
  
  parameters passed directly to `rapport`

**See Also**

`rapport.export` `rapport.html` `rapport.pdf` `rapport.odt`
**rapport.example**

**Template Examples**

**Description**

Displays template examples defined in Example section. Handy to check out what template does and how it looks once it’s rendered. If multiple examples are available, and index argument is NULL, you will be prompted for input. If only one example is available in the header, user is not prompted for input action, and given template is evaluated automatically. At any time you can provide an integer vector with example indices to index argument, and specified examples will be evaluated without prompting, thus returning a list of rapport objects. Example output can be easily exported to various formats (HTML, ODT, etc.) - check out documentation for rapport.export for more info.

**Usage**

```r
rapport.example(fp, index = NULL, env = .GlobalEnv)
```

**Arguments**

- `fp` a template file pointer (see `rapport::rapport.read` for details)
- `index` a numeric vector indicating the example index - meaningful only for templates with multiple examples. Accepts vector of integers to match IDs of template example. Using 'all' (character string) as index will return all examples.
- `env` an environment where example will be evaluated (defaults to .GlobalEnv)

**Examples**

```r
## Not run:
rapport.example('Example')
rapport.example('Example', 1:2)
rapport.example('Example', 'all')
rapport.example('Crosstable')
rapport.export(rapport.example('Crosstable'))

## End(Not run)
```

**rapport.export**

**Export rapport object**

**Description**

This function exports rapport class objects to various formats based on the pander package.
Usage

```r
rapport.export(rp = NULL, file, append = FALSE, create = TRUE,
open = TRUE, date = pander_return(Sys.time()), description = TRUE,
format = "html", options = "", logo = TRUE)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rp</code></td>
<td>a rapport class object or list of rapport class objects</td>
</tr>
<tr>
<td><code>file</code></td>
<td>filename of the generated document. Inherited from rapport class if not set. If file is set with path (not equal to <code>getwd()</code>), please set an absolute path for images (see: <code>eval$options()</code>).</td>
</tr>
<tr>
<td><code>append</code></td>
<td>FALSE (new report created) or an R object (class of &quot;Report&quot;) to which the new report will be added</td>
</tr>
<tr>
<td><code>create</code></td>
<td>should export really happen? It might be handy if you want to append several reports.</td>
</tr>
<tr>
<td><code>open</code></td>
<td>open the exported document? Default set to TRUE.</td>
</tr>
<tr>
<td><code>date</code></td>
<td>character string as the date field of the report. If not set, current time will be set.</td>
</tr>
<tr>
<td><code>description</code></td>
<td>add Description of the rapport class (template)? Default set to TRUE.</td>
</tr>
<tr>
<td><code>format</code></td>
<td>format of the wanted report. See Pandoc’s user manual for details. In short, choose something like: html, pdf, odt or docx.</td>
</tr>
<tr>
<td><code>options</code></td>
<td>options passed to Pandoc.convert.</td>
</tr>
<tr>
<td><code>logo</code></td>
<td>add rapport logo</td>
</tr>
</tbody>
</table>

Details

By default this function tries to export the report to HTML with pandoc. Some default styles are applied. If you do not like those default settings, use your own options argument.

Default parameters are read from global options:

- ‘rapport.user’

Please be sure to set ‘rapport.user’ option with `options()` to get your name in the head of your generated reports!

Value

filepath on create = TRUE, Report class otherwise

References


See Also

`rapport.html`, `rapport.pdf`, `rapport.odt`, `rapport.docx`
Examples

## Not run:

```r
## eval some template
x <- rapport('Example', data = mtcars, var = "hp")

## try basic parameters
rapport.export(x)
rapport.export(x, file = 'demo')
rapport.export(x, file = 'demo', format = 'odt')

### append reports
# 1) Create a report object with the first report and do not export (optional)
report <- rapport.export(x, create = F)
# 2) Append some other reports without exporting (optional)
report <- rapport.export(x, create = F, append = report)
# 3) Export it!
rapport.export(append=report)
# 4) Export it to other formats too! (optional)
rapport.export(append=report, format='rst')

### exporting multiple reports at once
rapport.export(rapport.example('Example', 'all'))
rapport.export(rapport.example('Example', 'all'), format = 'odt')
rapport.export(list(rapport('univar-descriptive', data = mtcars, var = "hp"),
                     rapport('Descriptives', data = mtcars, var = "mpg")))

### Never do this as being dumb:
rapport.export()

### Adding own custom CSS to exported HTML
rapport.export(x, options = sprintf('-c %s', system.file('templates/css/default.css', package='rapport')))```

## End(Not run)

---

**rapport.header**

---

**Template Header**

---

**Description**

Returns `rapport` template header from provided path or a character vector.

**Usage**

```r
rapport.header(fp, open.tag = get.tags("header.open"),
               close.tag = get.tags("header.close"), ...)```
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp</td>
<td>a template file pointer (see rapport:::rapport.read for details)</td>
</tr>
<tr>
<td>open.tag</td>
<td>a string with opening tag (defaults to value of user-defined &quot;header.open&quot; tag)</td>
</tr>
<tr>
<td>close.tag</td>
<td>a string with closing tag (defaults to value of user-defined &quot;header.close&quot; tag)</td>
</tr>
<tr>
<td>...</td>
<td>additional arguments to be passed to grep function</td>
</tr>
</tbody>
</table>

Value

a character vector with template header contents

Description

This is a simple wrapper around rapport and rapport.export. Basically it works like rapport but the returned class is exported at one go.

Usage

rapport.html(...)

Arguments

... parameters passed directly to rapport

See Also

rapport.export rapport.pdf rapport.odt rapport.docx

Description

Provides information about template metadata and/or inputs. See rapport.meta and rapport.inputs for details.

Usage

rapport.info(fp, meta = TRUE, inputs = TRUE)
**Arguments**

- **fp**: a template file pointer (see `rapport::rapport.read` for details)
- **meta**: return template metadata? (defaults to TRUE)
- **inputs**: return template inputs? (defaults to TRUE)

**See Also**

`rapport.meta` `rapport.inputs`

**Examples**

```r
## Not run:
rapport.info('Example')  # return both metadata and inputs
rapport.info('Crosstable', inputs = FALSE)  # return only template metadata
rapport.info('Correlation', meta = FALSE)  # return only template inputs
```

## End(Not run)

---

**Description**

Displays summary for template inputs (if any). Note that as of version 0.5, `rapport` template inputs should be defined using YAML syntax. See `deprecated.inputs` for details on old input syntax. The following sections describe new YAML input definition style.

**Usage**

```r
rapport.inputs(fp, use.header = FALSE)
```

**Arguments**

- **fp**: a template file pointer (see `rapport::rapport.read` for details)
- **use.header**: a logical value indicating whether the header section is provided in `h` argument

**Details**

### Introduction

The full power of `rapport` comes into play with *template inputs*. One can match inputs against dataset variables or custom R objects. The inputs provide means of assigning R objects to symbols in the template evaluation environment. Inputs themselves do not handle only the template names, but also provide an extensive set of rules that each dataset variable/user-provided R object has to satisfy. The new YAML input specification takes advantage of R class system. The input attributes should resemble common R object attributes and methods.

Inputs can be divided into two categories:
• dataset inputs, i.e. the inputs that refer to named element of an `codeR` object provided in `data` argument in `rapport` call. Currently, `rapport` supports only `data.frame` objects, but that may change in the (near) future.

• standalone inputs - the inputs that do not depend on the dataset. The user can just provide an R object of an appropriate class (and other input attributes) to match a standalone input.

General input attributes

Following attributes are available for all inputs:

• name (character string, required) - input name. It acts as an identifier for a given input, and is required as such. Template cannot contain duplicate names, `rapport` inputs currently have custom naming conventions - see `guess.input.name` for details.

• label (character string) - input label. It can be blank, but it's useful to provide input label as rapport helpers use that information in plot labels and/or exported HTML tables. Defaults to empty string.

• description (character string) - similar to label, but should contain long description of given input.

• class (character string) - defines an input class. Currently supported input classes are: character, complex, factor, integer, logical, numeric and raw (all atomic vector classes are supported). Class attribute should usually be provided, but it can also be NULL (default) - in that case the input class will be guessed based on matched R object's value.

• required (logical value) - does the input require a value? Defaults to FALSE.

• standalone (logical value) - indicates that the input depends on a dataset. Defaults to FALSE.

• length (either an integer value or a named list with integer values) - provides a set of rules for input value's length. Length attribute can be defined via:
  – an integer value, e.g. length: 10, which sets restriction to exactly 10 vectors or values.
  – named list with `min` and/or `max` attributes nested under `length` attribute. This will define a range of values in which input length must fall. Note that range limits are inclusive. Either `min` or `max` attribute can be omitted, and they will default to 1 and `Inf`, respectively.

IMPORTANT! Note that `rapport` treats input length in a bit different manner. If you match a subset of 10 character vectors from the dataset, input length will be 10, as you might expect. But if you select only one variable, length will be equal to 1, and not to the number of vector elements. This stands both for standalone and dataset inputs. However, if you match a character vector against a standalone input, length will be stored correctly - as the number of vector elements.

• value (a vector of an appropriate class). This attribute only exists for standalone inputs. Provided value must satisfy rules defined in `class` and `length` attributes, as well as any other class-specific rules (see below).

Class-specific attributes

class

• nchar - restricts the number of characters of the input value. It accepts the same attribute format as `length`. If `NULL` (default), no checks will be performed.
• **regexp** (character string) - contains a string with regular expression. If non-NULL, all strings in a character vector must match the given regular expression. Defaults to NULL - no checks are applied.

• **matchable** (logical value) - if TRUE, options attribute must be provided, while value is optional, though recommended. options should contain values to be chosen from, just like `<option>` tag does when nested in `<select>` HTML tag, while value must contain a value from options or it can be omitted (NULL). allow_multiple will allow values from options list to be matched multiple times. Note that unlike previous versions of rapport, partial matching is not performed.

**numeric, integer**

• **limit** - similar to length attribute, but allows only min and max nested attributes. Unlike length attribute, limit checks input values rather than input length. limit attribute is NULL by default and the checks are performed only when limit is defined (non-NULL).

**factor**

• **nlevels** - accepts the same format as length attribute, but the check is performed rather on the number of factor levels.

• **matchable** - *ibid* as in character inputs (note that in previous versions of rapport matching was performed against factor levels - well, not any more, now we match against values to make it consistent with character inputs).

**See Also**

`rapport.meta` `rapport.info`

---

**Description**

Lists all templates bundled with current package build. By default, it will search for all `.rapport` files in current directory, path specified in `rapport.paths` option and package library path.

**Usage**

`rapport.ls(...)`

**Arguments**

... additional parameters for `dir` function

**Value**

a character vector with template files
**Description**

Displays summary of template metadata stored in a header section. This part of template header consists of several YAML key: value pairs, which contain some basic information about the template, just much like the DESCRIPTION file in R packages does.

**Usage**

```r
rapport.meta(fp, fields = NULL, use.header = FALSE, trim.white = TRUE)
```

**Arguments**

- `fp`  
  a template file pointer (see `rapport:::rapport.read` for details)

- `fields`  
  a list of named lists containing key-value pairs of field titles and corresponding regexes

- `use.header`  
  a logical value indicating if the character vector provided in `fp` argument contains only the header data (not the whole template)

- `trim.white`  
  a logical value indicating if the extra spaces should removed from header fields before extraction

**Details**

Current implementation supports following fields:

- **title** - a template title (required)
- **author** - author’s (nick)name (required)
- **description** - template description (required)
- **email** - author’s email address
- **packages** - YAML list of packages required by the template (if any)
- **example** - example calls to `rapport` function, including template data and inputs

As of version 0.5, `dataRequired` field is deprecated. `rapport` function will automatically detect if the template requires a dataset based on the presence of `standalone` inputs.

**Value**

a named list with template metadata

**See Also**

`rapport.inputs` `rapport.info`
**Description**

This is a simple wrapper around `rapport` and `rapport.export`. Basically it works like `rapport` but the returned class is exported at one go.

**Usage**

```
rapport.odt(...)  
```

**Arguments**

... parameters passed directly to `rapport`  

**See Also**

`rapport.export` `rapport.html` `rapport.pdf` `rapport.docx`

---

**Description**

List all custom paths where rapport will look for templates.

**Usage**

```
rapport.path()  
```

**Value**

a character vector with paths

**Examples**

```R
## Not run:
rapport.path()

## End(Not run)
```
### rapport.path.add  
*Add Template Path*

**Description**

Adds a new element to custom paths’ list where rapport will look for templates.

**Usage**

```r
rappor.path.add(...)```

**Arguments**

- `...` character vector of paths

**Value**

TRUE on success (invisibly)

**Examples**

```r
## Not run:
rappor.path.add('/tmp')
rappor.ls()

## End(Not run)
```

### rapport.path.remove  
*Remove Template Path*

**Description**

Removes an element from custom paths’ list where rapport will look for templates.

**Usage**

```r
rappor.path.remove(...)```

**Arguments**

- `...` character vector of paths

**Value**

TRUE on success (invisibly)
Examples

```r
## Not run:
rapport.path()
rapport.path.add('/tmp')
rapport.path()
rapport.path.remove('/tmp')
rapport.path()

## End(Not run)
```

---

**rapport.path.reset**

*Reset Template Paths*

**Description**

Resets to default (NULL) all custom paths where rapport will look for templates.

**Usage**

```r
rapport.path.reset()
```

**Examples**

```r
## Not run:
rapport.path.reset()

## End(Not run)
```

---

**rapport.pdf**

*Rapport to PDF*

**Description**

This is a simple wrapper around `rapport` and `rapport.export`. Basically it works like `rapport` but the returned class is exported at one go.

**Usage**

```r
rapport.pdf(...)```

**Arguments**

... parameters passed directly to `rapport`

**See Also**

`rapport.export` `rapport.html` `rapport.odt` `rapport.docx`
**rapport.read**  
*Read Template*

**Description**

Reads file either from template name in system folder, file path (see `rapport.path`) or remote URL, and splits it into lines for easier handling by `rapport` internal parser.

**Usage**

```
rapport.read(fp, ...)  
```

**Arguments**

- `fp`: a character string containing a template path, a template name (for package-bundled templates only), template contents separated by newline (\n), or a character vector with template contents.
- `...`: additional params for header tag matching (see `grep`)

**Value**

- a character vector with template contents

**rapport.renew**  
*Renew deprecated template*

**Description**

Convert old-style template to new-style one (what we really do is just replacing old header syntax with YAML one).

**Usage**

```
rapport.renew(fp, file = NULL)  
```

**Arguments**

- `fp`: pointer to an old template (see `rapport::rapport.read` for details)
- `file`: a path to output file. If `NULL`, result will be flushed to stdout.
**rapport.rerun**

*Reproduce Template*

**Description**

Runs template with data and arguments included in `rapport` object. In order to get reproducible example, you have to make sure that `reproducible` argument is set to `TRUE` in `rapport` function.

**Usage**

```r
rapport.rerun(tpl)
```

**Arguments**

- **tpl**
  - A `rapport` object

**Examples**

```r
## Not run:
tmp <- rapport("Example", mtcars, v = "hp", reproducible = TRUE)
rapport.rerun(tmp)

## End(Not run)
```

---

**rapport.tangle**

*Extract template chunk contents*

**Description**

`rapport`'s alternative to `stangle` - extracts contents of template chunks. If `file` argument

**Usage**

```r
rapport.tangle(fp, file = "", show.inline.chunks = FALSE)
```

**Arguments**

- **fp**
  - Template file pointer (see `rapport:::rapport.read` for details)
- **file**
  - See file argument in `cat` function documentation
- **show.inline.chunks**
  - Extract contents of inline chunks as well? (defaults to `FALSE`)

**Value**

(invisibly) a list with either inline or block chunk contents
Index

adj.rle, 20
alike.integer, 20
as.character.rapport.inputs, 4
as.character.rapport.meta, 4
as.yaml.bool, 5
capitalise, 20
cat, 35
catn, 20
check.input.value, 5
check.input.value.class, 6
check.report.chunks, 6
check.tpl, 7
dir, 29
eject.meta, 7
fml, 20
get.tags, 8
grep, 7, 21, 26, 34
guess.input, 9
guess.input.description, 9
guess.input.label, 10
guess.input.name, 10, 28
guess.i, 10
guess.old.input.length, 11
guess.old.input.type, 11
htest, 21
htest.short, 21
inputs-deprecated, 12
is.boolean, 20
is.empty, 20
is.number, 20
is.rapport, 14
is.string, 20
is.tabular, 20
is.variable, 20
ius2008, 14
kurtosis, 21
lambda.test, 21
match.arg, 13
messagef, 20
pct, 20
print.rapport, 16
print.rapport.info, 16
print.rapport.inputs, 17
print.rapport.meta, 17
rapport, 3, 4, 18, 22, 26, 31, 33
rapport.helpers, 19
rapport-package, 3
rapport.body, 21
rapport.check.template, 22
rapport.docx, 20, 22, 24, 26, 31, 33
rapport.example, 23
rapport.export, 3, 20, 22, 23, 26, 31, 33
rapport.header, 16, 25
rapport.html, 20, 22, 24, 26, 31, 33
rapport.info, 26, 29, 30
rapport.inputs, 17, 18, 26, 27, 27, 30
rapport.ls, 29
rapport.meta, 17, 26, 27, 29, 30
rapport.odt, 20, 22, 24, 26, 31, 33
rapport.path, 31
rapport.path.add, 32
rapport.path.remove, 32
rapport.path.reset, 33
rapport.pdf, 20, 22, 24, 26, 31, 33
rapport.read, 34
rapport.renew, 34
rapport.rerun, 18, 35
rapport.tangle, 35
rp.desc, 20
rp.freq, 20
rp.iqr, 20
rp.max, 20
rp.mean, 21
rp.median, 21
rp.min, 21
rp.missing, 21
rp.outlier, 21
rp.percent, 21
rp.range, 21
rp.sd, 21
rp.se.mean, 21
rp.sum, 21
rp.valid, 21
rp.var, 21

skewness, 21
Stangle, 35
stopf, 20

tempfile, 3, 18
tocamel, 20
tpl.body (rapport.body), 21
tpl.check (rapport.check.template), 22
tpl.example (rapport.example), 23
tpl.export (rapport.export), 23
tpl.find (rapport.read), 34
tpl.header (rapport.header), 25
tpl.info (rapport.info), 26
tpl.inputs (rapport.inputs), 27
tpl.list (rapport.ls), 29
tpl.meta (rapport.meta), 30
tpl.paths (rapport.path), 31
tpl.paths.add (rapport.path.add), 32
tpl.paths.remove (rapport.path.remove), 32
tpl.paths.reset (rapport.path.reset), 33
tpl.renew (rapport.renew), 34
tpl.rerun (rapport.rerun), 35
tpl.tangle (rapport.tangle), 35
trim.space, 20

vgsub, 20

warningf, 20