Package: pegboard (via r-universe)

August 24, 2024

Title Explore and Manipulate Markdown Curricula from the Carpentries

Version 0.7.6

Description The Carpentries (<https://carpentries.org>) curricula is made of of lessons that are hosted as websites. Each lesson represents between a half day to two days of instruction and contains several episodes, which are written as 'kramdown'-flavored 'markdown' documents and converted to HTML using the 'Jekyll' static website generator. This package builds on top of the 'tinkr' package; reads in these markdown documents to 'XML' and stores them in R6 classes for convenient exploration and manipulation of sections within episodes.

License MIT + file LICENSE

URL https://carpentries.github.io/pegboard

BugReports https://github.com/carpentries/pegboard/issues

Imports commonmark, fs (>= 1.5.0), glue, purrr, R6, tinkr (>= 0.2.0), xml2, xslt, yaml

Suggests cli (>= 0.3.4), covr, crayon, dplyr, gert (>= 1.0.0), here, knitr, magrittr, rlang, rmarkdown, testthat, withr

VignetteBuilder knitr

Remotes ropensci/tinkr

Additional_repositories https://carpentries.r-universe.dev/

Config/testthat/edition 3

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Repository https://carpentries.r-universe.dev

RemoteUrl https://github.com/carpentries/pegboard

RemoteRef 0.7.6

RemoteSha ad2542f7f7b9c3f90cc871b355ff81ec14a09ca9

Contents

`collect_labels` *Collect and append validation messages*

Description

Given a data frame containing the results of validation tests, this will append a column of labels that describes each failure.

Usage

```
collect_labels(VAL, cli = FALSE, msg = heading_tests)
```

Arguments

`VAL`	a data frame containing the results of tests
`cli`	indicator to use the cli package to format warnings
`msg`	a named vector of template messages to provide for each test formatted for the **glue** package.

See Also

`throw_link_warnings()` for details on how this is implemented.

Examples

```
# As an example, consider a data frame where you have observations in rows
# and the results of individual tests in columns:
set.seed(2023-11-16)
dat <- data.frame(
  name = letters[1:10],
  rank = sample(1:3, 10, replace = TRUE),
  A = sample(c(TRUE, FALSE), 10, replace = TRUE, prob = c(7, 3)),
  B = sample(c(TRUE, FALSE), 10, replace = TRUE, prob = c(7, 3)),
  C = sample(c(TRUE, FALSE), 10, replace = TRUE, prob = c(7, 3))
)
dat
# you can see what the results of the tests were, but it would be a good
# idea to have a lookup table describing what these results mean
dat_tests <- c(
  A = "[missing widget]: {name}",
  B = "[incorrect rank]: {rank}",
  C = "[something else]"
)
# collect_labels will create the output you need:
pb <- asNamespace("pegboard")
res <- pb$collect_labels(dat, msg = dat_tests)
res
writeLines(res$labels)
if (requireNamespace("cli", quietly = TRUE)) {
  # you can also specify cli to TRUE to format with CLI
  res <- pb$collect_labels(dat, cli = TRUE, msg = dat_tests)
  writeLines(res$labels)
}
```

`Episode` *Class representing XML source of a Carpentries episode*

Description

Wrapper around an xml document to manipulate and inspect Carpentries episodes

Details

The Episode class is a superclass of `tinkr::yarn()`, which transforms (commonmark-formatted) Markdown to XML and back again. The extension that the Episode class provides is support for both Pandoc and kramdown flavours of Markdown.

Read more about this class in `vignette("intro-episode", package = "pegboard")`.

Super class

`tinkr::yarn` -> `Episode`

Public fields

`children` [`character`] a vector of absolute paths to child files if they exist.

`parents` [`character`] a vector of absolute paths to immediate parent files if they exist

`build_parents` [`character`] a vector of absolute paths to the final parent files that will trigger this child file to build

Active bindings

`show_problems` [`list`] a list of all the problems that occurred in parsing the episode

`headings` [`xml_nodeset`] all headings in the document

`links` [`xml_nodeset`] all links (not images) in the document

`images` [`xml_nodeset`] all image sources in the document

`tags` [`xml_nodeset`] all the kramdown tags from the episode

`questions` [`character`] the questions from the episode

`keypoints` [`character`] the keypoints from the episode

`objectives` [`character`] the objectives from the episode

`challenges` [`xml_nodeset`] all the challenges blocks from the episode

`solutions` [`xml_nodeset`] all the solutions blocks from the episode

`output` [`xml_nodeset`] all the output blocks from the episode

`error` [`xml_nodeset`] all the error blocks from the episode

`warning` [`xml_nodeset`] all the warning blocks from the episode

`code` [`xml_nodeset`] all the code blocks from the episode

`name` [`character`] the name of the source file without the path

`lesson` [`character`] the path to the lesson where the episode is from

`has_children` [`logical`] an indicator of the presence of child files (`TRUE`) or their absence (`FALSE`)

`has_parents` [`logical`] an indicator of the presence of parent files (`TRUE`) or their absence (`FALSE`)

Methods

Public methods:

- `Episode$new()`
- `Episode$confirm_sandpaper()`
- `Episode$get_blocks()`
- `Episode$get_images()`
- `Episode$label_divs()`
- `Episode$get_divs()`
- `Episode$get_yaml()`
- `Episode$use_dovetail()`
- `Episode$use_sandpaper()`
- `Episode$remove_error()`
- `Episode$remove_output()`

- `Episode$move_objectives()`
- `Episode$move_keypoints()`
- `Episode$move_questions()`
- `Episode$get_challenge_graph()`
- `Episode$show()`
- `Episode$head()`
- `Episode$tail()`
- `Episode$write()`
- `Episode$handout()`
- `Episode$reset()`
- `Episode$isolate_blocks()`
- `Episode$unblock()`
- `Episode$summary()`
- `Episode$validate_headings()`
- `Episode$validate_divs()`
- `Episode$validate_links()`
- `Episode$clone()`

Method `new()`**:** Create a new Episode

Usage:

```
Episode$new(
  path = NULL,
  process_tags = TRUE,
  fix_links = TRUE,
  fix_liquid = FALSE,
  parents = NULL,
  ...
)
```

Arguments:

`path` `[character]` path to a markdown episode file on disk

`process_tags` `[logical]` if `TRUE` (default), kramdown tags will be processed into attributes of the parent nodes. If `FALSE`, these tags will be treated as text

`fix_links` `[logical]` if `TRUE` (default), links pointing to liquid tags (e.g. `{{ page.root }}`) and included links (those supplied by a call to `{\% import links.md \%}`) will be appropriately processed as valid links.

`fix_liquid` `[logical]` defaults to `FALSE`, which means data is immediately passed to tinkr::yarn. If `TRUE`, all liquid variables in relative links have spaces removed to allow the commonmark parser to interpret them as links.

`parents` `[list]` a list of `Episode` objects that represent the immediate parents of this child

`...` arguments passed on to tinkr::yarn and `tinkr::to_xml()`

Returns: A new Episode object with extracted XML data

Examples:

```
scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
scope$name
scope$lesson
scope$challenges
```

Method `confirm_sandpaper()`**:** enforce that the episode is a sandpaper episode withtout going through the conversion steps. The default Episodes from pegboard were assumed to be generated using Jekyll with kramdown syntax. This is a bit of a kludge to bypass the normal checks for kramdown syntax and just assume pandoc syntax

Usage:

```
Episode$confirm_sandpaper()
```

Method `get_blocks()`**:** return all `block_quote` elements within the Episode

Usage:

```
Episode$get_blocks(type = NULL, level = 1L)
```

Arguments:

`type` the type of block quote in the Jekyll syntax like ".challenge", ".discussion", or ".solution"

`level` the level of the block within the document. Defaults to `1`, which represents all of the block_quotes are not nested within any other block quotes. Increase the nubmer to increase the level of nesting.

Returns: `[xml_nodeset]` all the blocks from the episode with the given tag and level.

Examples:

```
scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
# get all the challenges
scope$get_blocks(".challenge")
# get the solutions
scope$get_blocks(".solution", level = 2)
\dontrun{

  # download the source files for r-novice-gampinder into a Lesson object
  rng <- get_lesson("swcarpentry/r-novice-gapminder")
  dsp1 <- rng$episodes[["04-data-structures-part1.md"]]
  # There are 9 blocks in total
  dsp1$get_blocks()
  # One is a callout block
  dsp1$get_blocks(".callout")
  # One is a discussion block
  dsp1$get_blocks(".discussion")
  # Seven are Challenge blocks
  dsp1$get_blocks(".challenge")
  # There are eight solution blocks:
  dsp1$get_blocks(".solution", level = 2L)
}
```

Method `get_images()`**:** fetch the image sources and optionally process them for easier parsing. The default version of this function is equivalent to the active binding `$images`.

Usage:

```
Episode$get_images(process = FALSE)
```

Arguments:

`process` if `TRUE`, images will be processed via the internal function `process_images()`, which will add the `alt` attribute, if available and extract img nodes from HTML blocks.

Returns: an `xml_nodelist`

Examples:

```
loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
loop$get_images()
loop$get_images(process = TRUE)
```

Method `label_divs()`**:** label all the div elements within the Episode to extract them with `$get_divs()`

Usage:

```
Episode$label_divs()
```

Method `get_divs()`**:** return all div elements within the Episode

Usage:

```
Episode$get_divs(type = NULL, include = FALSE)
```

Arguments:

`type` the type of div tag (e.g. 'challenge' or 'solution')

`include` `\[logical\]` if `TRUE`, the div tags will be included in the output. Defaults to `FALSE`, which will only return the text between the div tags.

Method `get_yaml()`**:** Extract the yaml metadata from the episode

Usage:

```
Episode$get_yaml()
```

Method `use_dovetail()`**:** Ammend or add a setup code block to use `{dovetail}`

This will convert your lesson to use the dovetail R package for processing specialized block quotes which will do two things:

1. convert your lesson from md to Rmd
2. add to your setup chunk the following code

   ```
   library('dovetail')
   source(dvt_opts())
   ```

If there is no setup chunk, one will be created. If there is a setup chunk, then the `source` and `knitr_fig_path` calls will be removed.

Usage:

```
Episode$use_dovetail()
```

Method `use_sandpaper()`**:** Use the sandpaper package for processing

This will convert your lesson to use the `{sandpaper}` R package for processing the lesson instead of Jekyll (default). Doing this will have the following effects:

1. code blocks that were marked with liquid tags (e.g. `{: .language-r}` are converted to standard code blocks or Rmarkdown chunks (with language information at the top of the code block)
2. If rmarkdown is used and the lesson contains python code, `library('reticulate')` will be added to the setup chunk of the lesson.

Usage:

```
Episode$use_sandpaper(rmd = FALSE, yml = list())
```

Arguments:

`rmd` if `TRUE`, lessons will be converted to RMarkdown documents

`yml` the list derived from the yml file for the episode

Method `remove_error()`**:** Remove error blocks

Usage:

```
Episode$remove_error()
```

Method `remove_output()`**:** Remove output blocks

Usage:

```
Episode$remove_output()
```

Method `move_objectives()`**:** move the objectives yaml item to the body

Usage:

```
Episode$move_objectives()
```

Method `move_keypoints()`**:** move the keypoints yaml item to the body

Usage:

```
Episode$move_keypoints()
```

Method `move_questions()`**:** move the questions yaml item to the body

Usage:

```
Episode$move_questions()
```

Method `get_challenge_graph()`**:** Create a graph of the top-level elements for the challenges.

Usage:

```
Episode$get_challenge_graph(recurse = TRUE)
```

Arguments:

`recurse` if `TRUE` (default), the content of the solutions will be included in the graph; `FALSE` will keep the solutions as `block_quote` elements.

Returns: a data frame with four columns representing all the elements within the challenges in the Episode:

- Block: The sequential number of the challenge block
- from: the inward elements
- to: the outward elements
- pos: the position in the markdown document

Note that there are three special node names:

- challenge: start or end of the challenge block
- solution: start of the solution block
- lesson: start of the lesson block

Examples:

```
scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
scope$get_challenge_graph()
```

Method `show()`**:** show the markdown contents on the screen

Usage:

```
Episode$show()
```

Returns: a character vector with one line for each line of output

Examples:

```
scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
scope$head()
scope$tail()
scope$show()
```

Method `head()`**:** show the first n lines of markdown contents on the screen

Usage:

```
Episode$head(n = 6L)
```

Arguments:

`n` the number of lines to show from the top

Returns: a character vector with one line for each line of output

Method `tail()`**:** show the first n lines of markdown contents on the screen

Usage:

```
Episode$tail(n = 6L)
```

Arguments:

`n` the number of lines to show from the top

Returns: a character vector with one line for each line of output

Method `write()`**:** write the episode to disk as markdown

Usage:

```
Episode$write(path = NULL, format = "md", edit = FALSE)
```

Arguments:

`path` the path to write your file to. Defaults to an empty directory in your temporary folder

`format` one of "md" (default) or "xml". This will create a file with the correct extension in the path

`edit` if `TRUE`, the file will open in an editor. Defaults to `FALSE`.

Returns: the episode object

Examples:

```
scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
scope$write()
```

Method `handout()`**:** Create a trimmed-down RMarkdown document that strips prose and contains only important code chunks and challenge blocks without solutions.

Usage:

```
Episode$handout(path = NULL, solutions = FALSE)
```

Arguments:

`path` (handout) a path to an R Markdown file to write. If this is `NULL`, no file will be written and the lines of the output will be returned.

`solutions` if `TRUE`, include solutions in the output. Defaults to `FALSE`, which removes the solution blocks.

Returns: a character vector if `path` = `NULL`, otherwise, it is called for the side effect of creating a file.

Examples:

```
lsn <- Lesson$new(lesson_fragment("sandpaper-fragment"), jekyll = FALSE)
e <- lsn$episodes[[1]]
cat(e$handout())
cat(e$handout(solution = TRUE))
```

Method `reset()`**:** Re-read episode from disk

Usage:

```
Episode$reset()
```

Returns: the episode object

Examples:

```
scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
xml2::xml_text(scope$tags[1])
xml2::xml_set_text(scope$tags[1], "{: .code}")
xml2::xml_text(scope$tags[1])
scope$reset()
xml2::xml_text(scope$tags[1])
```

Method `isolate_blocks()`**:** Remove all elements except for those within block quotes that have a kramdown tag. Note that this is a destructive process.

Usage:

```
Episode$isolate_blocks()
```

Returns: the Episode object, invisibly

Examples:

```
scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
scope$body # a full document with block quotes and code blocks, etc
scope$isolate_blocks()$body # only one challenge block_quote
```

Method `unblock()`**:** convert challenge blocks to roxygen-like code blocks

Usage:

```
Episode$unblock(token = "#'", force = FALSE)
```

Arguments:

`token` the token to use to indicate non-code, Defaults to "#'"

`force` force the conversion even if the conversion has already taken place

Returns: the Episode object, invisibly

Examples:

```
loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
loop$body # a full document with block quotes and code blocks, etc
loop$get_blocks() # all the blocks in the episode
loop$unblock()
loop$get_blocks() # no blocks
loop$code # now there are two blocks with challenge tags
```

Method `summary()`**:** Get a high-level summary of the elements in the episode

Usage:

```
Episode$summary()
```

Returns: a data frame with counts of the following elements per page:

- sections: level 2 headings
- headings: all headings
- callouts: all callouts
- challenges: subset of callouts
- solutions: subset of callouts
- code: all code block elements (excluding inline code)
- output: subset of code that is displayed as output
- warnining: subset of code that is displayed as a warning
- error: subset of code that is displayed as an error
- images: all images in markdown or HTML
- links: all links in markdown or HTML

Method `validate_headings()`**:** perform validation on headings in a document.

This will validate the following aspects of all headings:

- first heading starts at level 2 (`first_heading_is_second_level`)
- greater than level 1 (`greater_than_first_level`)
- increse sequentially (e.g. no jumps from 2 to 4) (`are_sequential`)
- have names (`have_names`)
- unique in their own hierarchy (`are_unique`)

Usage:

```
Episode$validate_headings(verbose = TRUE, warn = TRUE)
```

Arguments:

`verbose` if `TRUE` (default), a message for each rule broken will be issued to the stderr. if `FALSE`, this will be silent.

`warn` if `TRUE` (default), a warning will be issued if there are any failures in the tests.

Returns: a data frame with a variable number of rows and the follwoing columns:

- **episode** the filename of the episode
- **heading** the text from a heading
- **level** the heading level
- **pos** the position of the heading in the document
- **node** the XML node that represents the heading
- (the next five columns are the tests listed above)
- **path** the path to the file.

Each row in the data frame represents an individual heading across the Lesson. See `validate_headings()` for more details.

Examples:

```
# Example: There are multiple headings called "Solution" that are not
# nested within a higher-level heading and will throw an error
loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
loop$validate_headings()
```

Method `validate_divs():` perform validation on divs in a document.

This will validate the following aspects of divs. See `validate_divs()` for details.

- divs are of a known type (`is_known`)

Usage:

```
Episode$validate_divs(warn = TRUE)
```

Arguments:

`warn` if `TRUE` (default), a warning message will be if there are any divs determined to be invalid. Set to `FALSE` if you want the table for processing later.

Returns: a logical `TRUE` for valid divs and `FALSE` for invalid divs.

Examples:

```
loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
loop$validate_divs()
```

Method `validate_links():` perform validation on links and images in a document.

This will validate the following aspects of links. See `validate_links()` for details.

- External links use HTTPS (`enforce_https`)
- Internal links exist (`internal_okay`)
- External links are reachable (`all_reachable`) (planned)
- Images have alt text (`img_alt_text`)
- Link text is descriptive (`descriptive`)
- Link text is more than a single letter (`link_length`)

Usage:

```
Episode$validate_links(warn = TRUE)
```

Arguments:

`warn` if `TRUE` (default), a warning message will be if there are any links determined to be invalid. Set to `FALSE` if you want the table for processing later.

Returns: a logical `TRUE` for valid links and `FALSE` for invalid links.

Examples:

```
loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
loop$validate_links()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Episode$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

The current XLST spec for tinkr does not support kramdown, which the Carpentries Episodes are styled with, thus some block tags will be destructively modified in the conversion.

Examples

```
## ------------------------------------------------
## Method `Episode$new`
## ------------------------------------------------

scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
scope$name
scope$lesson
scope$challenges

## ------------------------------------------------
## Method `Episode$get_blocks`
## ------------------------------------------------

scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
# get all the challenges
scope$get_blocks(".challenge")
# get the solutions
scope$get_blocks(".solution", level = 2)
## Not run:

  # download the source files for r-novice-gampinder into a Lesson object
  rng <- get_lesson("swcarpentry/r-novice-gapminder")
  dsp1 <- rng$episodes[["04-data-structures-part1.md"]]
  # There are 9 blocks in total
  dsp1$get_blocks()
  # One is a callout block
  dsp1$get_blocks(".callout")
  # One is a discussion block
  dsp1$get_blocks(".discussion")
  # Seven are Challenge blocks
  dsp1$get_blocks(".challenge")
  # There are eight solution blocks:
```

```
  dsp1$get_blocks(".solution", level = 2L)

## End(Not run)

## ------------------------------------------------
## Method `Episode$get_images`
## ------------------------------------------------


loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
loop$get_images()
loop$get_images(process = TRUE)

## ------------------------------------------------
## Method `Episode$get_challenge_graph`
## ------------------------------------------------

scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
scope$get_challenge_graph()

## ------------------------------------------------
## Method `Episode$show`
## ------------------------------------------------

scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
scope$head()
scope$tail()
scope$show()

## ------------------------------------------------
## Method `Episode$write`
## ------------------------------------------------

scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
scope$write()

## ------------------------------------------------
## Method `Episode$handout`
## ------------------------------------------------

lsn <- Lesson$new(lesson_fragment("sandpaper-fragment"), jekyll = FALSE)
e <- lsn$episodes[[1]]
cat(e$handout())
cat(e$handout(solution = TRUE))

## ------------------------------------------------
## Method `Episode$reset`
## ------------------------------------------------

scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
xml2::xml_text(scope$tags[1])
xml2::xml_set_text(scope$tags[1], "{: .code}")
xml2::xml_text(scope$tags[1])
```

```
scope$reset()
xml2::xml_text(scope$tags[1])

## ------------------------------------------------
## Method `Episode$isolate_blocks`
## ------------------------------------------------

scope <- Episode$new(file.path(lesson_fragment(), "_episodes", "17-scope.md"))
scope$body # a full document with block quotes and code blocks, etc
scope$isolate_blocks()$body # only one challenge block_quote

## ------------------------------------------------
## Method `Episode$unblock`
## ------------------------------------------------

loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
loop$body # a full document with block quotes and code blocks, etc
loop$get_blocks() # all the blocks in the episode
loop$unblock()
loop$get_blocks() # no blocks
loop$code # now there are two blocks with challenge tags

## ------------------------------------------------
## Method `Episode$validate_headings`
## ------------------------------------------------

# Example: There are multiple headings called "Solution" that are not
# nested within a higher-level heading and will throw an error
loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
loop$validate_headings()

## ------------------------------------------------
## Method `Episode$validate_divs`
## ------------------------------------------------

loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
loop$validate_divs()

## ------------------------------------------------
## Method `Episode$validate_links`
## ------------------------------------------------

loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
loop$validate_links()
```

fix_liquid_relative_link

Remove spaces in relative links with liquid variables

Description

Liquid has a syntax that wraps variables in double moustache braces that may or may not have spaces within the moustaches. For example, to get the link of the page root, you would use page.root to make it more readable. However, this violates the expectation of the commonmark parser and makes it think "oh, this is just ordinary text".

Usage

```
fix_liquid_relative_link(path, encoding = "UTF-8")
```

Arguments

`path`	path to an MD file
`encoding`	encoding of the text, defaults to UTF-8

Details

This function fixes the issue by removing the spaces within the braces.

`fix_sandpaper_links` *Fix relative and jekyll links to be compatible with sandpaper*

Description

This function will perform the transformation on three node types:

Usage

```
fix_sandpaper_links(body, yml = list(), path = NULL, known = NULL)
```

Arguments

`body`	an XML document
`yml`	the list of key/value pairs derived from the `_config.yml` file
`path`	the path to the current episode
`known`	a character vector of known episodes in the lesson, relative to the lesson root.

Details

- image
- link
- html_node

The transformation will be to remove relative paths ("../") and replace Jekyll templating (e.g. " page.root " and " site.swc_pages " with either nothing or the link to software carpentry, respectively.

Value

the body, invisibly

Note

This is absolutely NOT comprehensive and some links will fail to be converted. If this happens, please report an issue: https://github.com/carpentries/pegboard/issues/new/

Examples

```
loop <- fs::path(lesson_fragment(), "_episodes", "14-looping-data-sets.md")
e <- Episode$new(loop)
pegboard:::make_link_table(e)$orig
e$use_sandpaper()
pegboard:::make_link_table(e)$orig
```

---

get_blocks *Gather blocks from the XML body of a carpentries lesson*

Description

This will search an XML document for `block_quotes` with the specified type and level and extract them into a nodeset.

Usage

```
get_blocks(body, type = NULL, level = 0)
```

Arguments

`body`	the XML body of a carpentries lesson (an xml2 object)
`type`	the type of block quote in the Jekyll syntax like ".challenge", ".discussion", or ".solution"
`level`	the level of the block within the document. Defaults to `1`, which represents all of the block_quotes are not nested within any other block quotes. Increase the nubmer to increase the level of nesting.

Value

an xml nodeset object with each element representing a blockquote that matched the input criteria.

Note

At the moment, blocks are returned at the specified level. If you select `type = ".solution", level = 1`, you will receive blocks that *contain* solution blocks even though these blocks are almost always nested within other blocks.

Examples

```
frg <- Lesson$new(lesson_fragment())
# Find all the blocks in the
get_blocks(frg$episodes[["17-scope.md"]]$body)
```

---

get_challenges *Gather challenges from the XML body of a carpentries lesson*

Description

This will search an XML document for a challenge marker and extract all of the block quotes that are ancestral to that marker so that we can extract the challenge blockquotes from the carpentries lessons.

Usage

```
get_challenges(body, type = c("block", "div", "chunk"))
```

Arguments

`body`	the XML body of a carpentries lesson (an xml2 object)
`type`	the type of element containing the challenges "block" is the default and will search for all of the blockquotes with liquid/kramdown markup, "div" will search for all div tags with class of challenge, and "chunk" will search for all of code chunks with the engine of challenge.

Value

an xml object.

Examples

```
loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
get_challenges(loop$body, "block")
get_challenges(loop$unblock()$body, "div")
loop$reset()
get_challenges(loop$use_dovetail()$unblock()$body, "chunk")
```

`get_code` *Get code blocks from xml document*

Description

Get code blocks from xml document

Usage

```
get_code(body, type = ".language-", attr = "@ktag")
```

Arguments

`body`	an xml document from a jekyll site
`type`	a full or partial string of a code block attribute from Jekyll without parenthesis.
`attr`	what attribute to query in search of code blocks. Default is @ktag, which will search for "{: \<type\>".

Details

This uses the XPath function `fn:starts-with()` to search for the code block and automatically includes the opening brace, so regular expressions are not allowed. This is used by the `$code`, `$output`, and `$error` elements of the Episode class.

Value

an xml nodeset object

Examples

```
e <- Episode$new(fs::path(lesson_fragment(), "_episodes", "17-scope.md"))

get_code(e$body)
get_code(e$body, ".output")
get_code(e$body, ".error")
```

`get_headings` *Get all headings in the XML document*

Description

Get all headings in the XML document

Usage

```
get_headings(body)

show_heading_tree(tree)
```

Arguments

`body`	an XML document
`tree`	a data frame produced via `validate_headings()`

Value

an object of class `xml_nodeset` with all the headings in the document.

`get_lesson` *Get a carpentries lesson in XML format*

Description

Download and extract a carpentries lesson in XML format. This uses `gert::git_clone()` to download a carpentries lesson to your computer (defaults to the temporary directory and extracts the lesson in `_episodes/` using `tinkr::to_xml()`

Usage

```
get_lesson(lesson = NULL, path = tempdir(), overwrite = FALSE, ...)
```

Arguments

`lesson`	a github user/repo pattern to point to the lesson
`path`	a directory to write the lesson to
`overwrite`	if the `path` exists, setting this to `TRUE` will overwrite the path, otherwise, the contents of the path will be returned if it is a lesson repository.
`...`	arguments passed on to Episode$new().

Value

a list of xml objects, one element per episode.

Examples

```
if (interactive()) {
  png <- get_lesson("swcarpentry/python-novice-gapminder")
  str(png, max.level = 1)
}
```

get_solutions *Gather solutions from the XML body of a carpentries lesson*

Description

This will search an XML document for a solution marker and extract all of the block quotes that are ancestral to that marker so that we can extract the solution blockquotes from the carpentries lessons.

Usage

```
get_solutions(body, type = c("block", "div", "chunk"), parent = NULL)
```

Arguments

`body`	the XML body of a carpentries lesson (an xml2 object)
`type`	the type of element containing the solutions "block" is the default and will search for all of the blockquotes with liquid/kramdown markup, "div" will search for all div tags with class of solution, and "chunk" will search for all of code chunks with the engine of solution.
`parent`	the outer block containing the solution. Default is a challenge block, but it could also be a discussion block.

Value

- type = "block" (default) an xml nodelist of blockquotes
- type = "div" a list of xml nodelists
- type = "chunk" an xml nodelist of code blocks

Note

- the `parent` parameter is only valid for the "block" (default) type
- the "chunk" type has the limitation that solutions are embedded within their respective blocks, so counting the number of solution elements via this method may an undercount

Examples

```
loop <- Episode$new(file.path(lesson_fragment(), "_episodes", "14-looping-data-sets.md"))
get_solutions(loop$body, "block")
get_solutions(loop$unblock()$body, "div")
loop$reset()
get_solutions(loop$use_dovetail()$unblock()$body, "chunk")
```

`isolate_elements` *Isolate elements in an XML document by source position*

Description

Isolate elements in an XML document by source position

Usage

```
isolate_elements(body, ...)
```

Arguments

`body`	an XML document
`...`	objects of class `xml_node` or `xml_nodeset` to be retained

Value

This works by side-effect, but it returns the body, invisibly.

`issue_warning` *Issue a warning via CLI if it exists or send a message*

Description

This allows us to control the messages emitted *and* continue to keep CLI as a suggested package.

Usage

```
issue_warning(
  msg = NULL,
  cli = has_cli(),
  what = NULL,
  url = NULL,
  n = NULL,
  N = NULL,
  infos = list(),
  reports = list(),
  ...
)

pb_message(..., domain = NULL, appendLF = TRUE)

line_report(msg = "", path, pos, sep = "\t", type = "warning")
```

```
append_labels(l, i = TRUE, e = "", cli = FALSE, f = "style_inverse")

message_muffler(expr, keep = FALSE)
```

Arguments

`msg`	the message as a glue or CLI string. Defaults to NULL
`cli`	if `TRUE`, stylizes `e` with `f`
`what`	the name of the specific element to report in an error
`url`	a url for extra information to help.
`n`	the number of elements errored
`N`	the number total elements
`infos`	the information about the errors to be shown to the user
`reports`	the reported errors.
`...`	named arguments to be evaluated in the message via glue or CLI
`domain`	see `gettext`. If `NA`, messages will not be translated, see also the note in `stop`.
`appendLF`	logical: should messages given as a character string have a newline appended?
`path`	path to the file to report
`pos`	position of the error
`sep`	a character to use to separate the human message and the line number
`type`	(used in the context of CI only) the type of warning that should be thrown (defaults to warning)
`l`	a vector/list of characters
`i`	the index of elements to append
`e`	the new element to append to each element
`f`	a function from **cli** that will transform `e`
`expr`	an R expression.
`keep`	if `TRUE`, the messages are kept in a list. Defautls to `FALSE` where cli message are discarded.

Details

The vast majority of the code in this function is copied directly from the `message()` function.

Value

nothing, invisibly; used for side-effect

, `l`, appended

if `keep = FALSE`, the output of `expr`, if `keep = TRUE`, a list with the elements `val = expr` and `msg = <cliMessage>s`

Examples

```
pegboard:::pb_message("hello")
x <- letters[1:5]
x2 <- pegboard:::append_labels(x,
  c(1, 3),
  "appended",
  cli = requireNamespace("cli", quietly = TRUE),
  f = "col_cyan"
)
writeLines(glue::glue("[{x}]->[{x2}]"))
pegboard:::message_muffler({
  cli::cli_text("hello there! I'm staying in!")
  pegboard:::pb_message("normal looking message that's not getting through")
  message("this message makes it out!")
  runif(1)
})
pegboard:::message_muffler({
  cli::cli_text("hello there! I'm staying in!")
  pegboard:::pb_message("normal looking message that's not getting through")
  message("this message makes it out!")
  runif(1)
}, keep = TRUE)
```

`Lesson` *Class to contain a single Lesson by the Carpentries*

Description

This is a wrapper for several Episode class objects.

Details

This class contains and keeps track of relationships between Episode objects contained within Carpentries Workbench and Carpentries styles lessons.

Read more about how to use this class in `vignette("intro-lesson", package = "pegboard")`

Public fields

`path [character]` path to Lesson directory

`episodes [list]` list of Episode class objects representing the episodes of the lesson.

`built [list]` list of Episode class objects representing the markdown artefacts rendered from RMarkdown files.

`extra [list]` list of Episode class objects representing the extra markdown components including index, setup, information for learners, information for instructors, and learner profiles. This is not processed for the jekyll lessons.

`children [list]` list of Episode class objects representing child files that are needed by any of the components to be built This is not processed for the jekyll lessons.

`sandpaper` `[logical]` when `TRUE`, the episodes in the lesson are written in pandoc flavoured markdown. `FALSE` would indicate a jekyll-based lesson written in kramdown.

`rmd` `[logical]` when `TRUE`, the episodes represent RMarkdown files, default is `FALSE` for markdown files (deprecated and unused).

`overview` `[logical]` when `TRUE`, the lesson is an overview lesson and does not necessarily contain any episodes. Defaults to `FALSE`

Active bindings

`n_problems` number of problems per episode

`show_problems` contents of the problems per episode

`files` the source files for each episode

`has_children` a logical indicating the presence (`TRUE`) or absence (`FALSE`) of child files within the main files of the lesson

Methods

Public methods:

- `Lesson$new()`
- `Lesson$load_built()`
- `Lesson$get()`
- `Lesson$summary()`
- `Lesson$blocks()`
- `Lesson$challenges()`
- `Lesson$solutions()`
- `Lesson$thin()`
- `Lesson$reset()`
- `Lesson$isolate_blocks()`
- `Lesson$handout()`
- `Lesson$validate_headings()`
- `Lesson$validate_divs()`
- `Lesson$validate_links()`
- `Lesson$trace_lineage()`
- `Lesson$clone()`

Method `new()`**:** create a new Lesson object from a directory

Usage:

```
Lesson$new(path = ".", rmd = FALSE, jekyll = TRUE, ...)
```

Arguments:

`path` `[character]` path to a lesson directory. This must have a folder called `_episodes` within that contains markdown episodes. Defaults to the current working directory.

`rmd` `[logical]` when `TRUE`, the imported files will be the source RMarkdown files. Defaults to `FALSE`, which reads the rendered markdown files.

`jekyll` `[logical]` when `TRUE` (default), the structure of the lesson is assumed to be derived from the carpentries/styles repository. When `FALSE`, The structure is assumed to be a sandpaper lesson and extra content for learners, instructors, and profiles will be populated.

`...` arguments passed on to Episode$new

Returns: a new Lesson object that contains a list of Episode objects in `$episodes`

Examples:

```
frg <- Lesson$new(lesson_fragment())
frg$path
frg$episodes
```

Method `load_built()`**:** read in the markdown content generated from RMarkdown sources and load load them into memory

Usage:

```
Lesson$load_built()
```

Method `get()`**:** A getter for various active bindings in the Episode class of objects. In practice this is syntactic sugar around `purrr::map(l$episodes, ~.x$element)`

Usage:

```
Lesson$get(element = NULL, collection = "episodes")
```

Arguments:

`element` `[character]` a defined element from the active bindings in the Episode class. Defaults to NULL, which will return nothing. Elements that do not exist in the Episode class will return NULL

`collection` `[character]` one or more of "episodes" (default), "extra", or "built". Select `TRUE` to collect information from all files.

Examples:

```
frg <- Lesson$new(lesson_fragment())
frg$get("error") # error code blocks
frg$get("links") # links
```

Method `summary()`**:** summary of element counts in each episode. This can be useful for assessing a broad overview of the lesson dynamics

Usage:

```
Lesson$summary(collection = "episodes")
```

Arguments:

`collection` `[character]` one or more of "episodes" (default), "extra", or "built". Select `TRUE` to collect information from all files.

Examples:

```
frg <- Lesson$new(lesson_fragment())
frg$summary() # episode summary (default)
```

Method `blocks()`**:** Gather all of the blocks from the lesson in a list of xml_nodeset objects

Usage:

```
Lesson$blocks(type = NULL, level = 0, path = FALSE)
```

Arguments:

`type` the type of block quote in the Jekyll syntax like ".challenge", ".discussion", or ".solution"

`level` the level of the block within the document. Defaults to `0`, which represents all of the block_quotes within the document regardless of nesting level.

`path` `[logical]` if `TRUE`, the names of each element will be equivalent to the path. The default is `FALSE`, which gives the name of each episode.

body the XML body of a carpentries lesson (an xml2 object)

Method `challenges()`**:** Gather all of the challenges from the lesson in a list of xml_nodeset objects

Usage:

```
Lesson$challenges(path = FALSE, graph = FALSE, recurse = TRUE)
```

Arguments:

`path` `[logical]` if `TRUE`, the names of each element will be equivalent to the path. The default is `FALSE`, which gives the name of each episode.

`graph` `[logical]` if `TRUE`, the output will be a data frame representing the directed graph of elements within the challenges. See the `get_challenge_graph()` method in Episode.

`recurse` `[logical]` when `graph = TRUE`, this will include the solutions in the output. See Episode for more details.

Method `solutions()`**:** Gather all of the solutions from the lesson in a list of xml_nodeset objects

Usage:

```
Lesson$solutions(path = FALSE)
```

Arguments:

`path` `[logical]` if `TRUE`, the names of each element will be equivalent to the path. The default is `FALSE`, which gives the name of each episode.

Method `thin()`**:** Remove episodes that have no challenges

Usage:

```
Lesson$thin(verbose = TRUE)
```

Arguments:

`verbose` `[logical]` if `TRUE` (default), the names of each episode removed is reported. Set to `FALSE` to remove this behavior.

Returns: the Lesson object, invisibly

Examples:

```
frg <- Lesson$new(lesson_fragment())
frg$thin()
```

Method `reset()`**:** Re-read all Episodes from disk

Usage:

```
Lesson$reset()
```

Returns: the Lesson object

Examples:

```
frg <- Lesson$new(lesson_fragment())
frg$episodes[[1]]$body
frg$isolate_blocks()$episodes[[1]]$body # empty
frg$reset()$episodes[[1]]$body # reset
```

Method `isolate_blocks()`**:** Remove all elements except for those within block quotes that have a kramdown tag. Note that this is a destructive process.

Usage:

```
Lesson$isolate_blocks()
```

Returns: the Episode object, invisibly

Examples:

```
frg <- Lesson$new(lesson_fragment())
frg$isolate_blocks()$body # only one challenge block_quote
```

Method `handout()`**:** create a handout for all episodes in the lesson

Usage:

```
Lesson$handout(path = NULL, solution = FALSE)
```

Arguments:

`path` the path to the R Markdown file to be written. If `NULL` (default), no file will be written and the lines of the output document will be returned.

`solution` if `TRUE` solutions will be retained. Defaults to `FALSE`

Returns: if `path = NULL`, a character vector, otherwise, the object itself is returned.

Examples:

```
lsn <- Lesson$new(lesson_fragment("sandpaper-fragment"), jekyll = FALSE)
cat(lsn$handout())
cat(lsn$handout(solution = TRUE))
```

Method `validate_headings()`**:** Validate that the heading elements meet minimum accessibility requirements. See the internal `validate_headings()` for deails.

This will validate the following aspects of all headings:

- first heading starts at level 2 (`first_heading_is_second_level`)
- greater than level 1 (`greater_than_first_level`)
- increse sequentially (e.g. no jumps from 2 to 4) (`are_sequential`)
- have names (`have_names`)
- unique in their own hierarchy (`are_unique`)

Usage:

```
Lesson$validate_headings(verbose = TRUE)
```

Arguments:

`verbose` if `TRUE`, the heading tree will be printed to the console with any warnings assocated with the validators

Returns: a data frame with a variable number of rows and the follwoing columns:

- **episode** the filename of the episode
- **heading** the text from a heading

- **level** the heading level
- **pos** the position of the heading in the document
- **node** the XML node that represents the heading
- (the next five columns are the tests listed above)
- **path** the path to the file.

Each row in the data frame represents an individual heading across the Lesson. See `validate_headings()` for more details.

Examples:

```
frg <- Lesson$new(lesson_fragment())
frg$validate_headings()
```

Method `validate_divs()`**:** Validate that the divs are known. See the internal `validate_divs()` for details.

Validation variables:

- divs are known (`is_known`)

Usage:

```
Lesson$validate_divs()
```

Arguments:

`verbose` if `TRUE` (default), Any failed tests will be printed to the console as a message giving information of where in the document the failing divs appear.

Returns: a wide data frame with five rows and the number of columns equal to the number of episodes in the lesson with an extra column indicating the type of validation. See the same method in the Episode class for details.

Examples:

```
frg <- Lesson$new(lesson_fragment())
frg$validate_divs()
```

Method `validate_links()`**:** Validate that the links and images are valid and accessible. See the internal `validate_links()` for details.

Validation variables:

- External links use HTTPS (`enforce_https`)
- Internal links exist (`internal_okay`)
- External links are reachable (`all_reachable`) (planned)
- Images have alt text (`img_alt_text`)
- Link text is descriptive (`descriptive`)
- Link text is more than a single letter (`link_length`)

Usage:

```
Lesson$validate_links()
```

Arguments:

`verbose` if `TRUE` (default), Any failed tests will be printed to the console as a message giving information of where in the document the failing links/images appear.

Returns: a wide data frame with five rows and the number of columns equal to the number of episodes in the lesson with an extra column indicating the type of validation. See the same method in the Episode class for details.

Examples:

```
frg <- Lesson$new(lesson_fragment())
frg$validate_links()
```

Method `trace_lineage()`**:** find all the children of a single source file

Usage:

```
Lesson$trace_lineage(episode_path)
```

Arguments:

`episode_path` the path to an episode or extra file

Returns: a character vector of the full lineage of files starting with a single source file. Note: this assumes a sandpaper lesson that has child files. If there are no child files, it will return the path

Examples:

```
frag <- lesson_fragment("sandpaper-fragment-with-child")
lsn <- Lesson$new(frag, jekyll = FALSE)
lsn$has_children # TRUE
lsn$episodes[[1]]$children # first episode shows 1 immediate child
lsn$trace_lineage(lsn$files[[1]]) # find recursive children of 1st episode
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Lesson$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `Lesson$new`
## ------------------------------------------------

frg <- Lesson$new(lesson_fragment())
frg$path
frg$episodes

## ------------------------------------------------
## Method `Lesson$get`
## ------------------------------------------------

frg <- Lesson$new(lesson_fragment())
frg$get("error") # error code blocks
frg$get("links") # links

## ------------------------------------------------
## Method `Lesson$summary`
## ------------------------------------------------

frg <- Lesson$new(lesson_fragment())
frg$summary() # episode summary (default)

## ------------------------------------------------
## Method `Lesson$thin`
## ------------------------------------------------

frg <- Lesson$new(lesson_fragment())
frg$thin()

## ------------------------------------------------
## Method `Lesson$reset`
## ------------------------------------------------

frg <- Lesson$new(lesson_fragment())
frg$episodes[[1]]$body
frg$isolate_blocks()$episodes[[1]]$body # empty
frg$reset()$episodes[[1]]$body # reset

## ------------------------------------------------
## Method `Lesson$isolate_blocks`
## ------------------------------------------------

frg <- Lesson$new(lesson_fragment())
frg$isolate_blocks()$body # only one challenge block_quote

## ------------------------------------------------
## Method `Lesson$handout`
## ------------------------------------------------

lsn <- Lesson$new(lesson_fragment("sandpaper-fragment"), jekyll = FALSE)
cat(lsn$handout())
cat(lsn$handout(solution = TRUE))

## ------------------------------------------------
## Method `Lesson$validate_headings`
## ------------------------------------------------

frg <- Lesson$new(lesson_fragment())
frg$validate_headings()

## ------------------------------------------------
## Method `Lesson$validate_divs`
## ------------------------------------------------

frg <- Lesson$new(lesson_fragment())
frg$validate_divs()

## ------------------------------------------------
## Method `Lesson$validate_links`
## ------------------------------------------------

frg <- Lesson$new(lesson_fragment())
frg$validate_links()

## ------------------------------------------------
## Method `Lesson$trace_lineage`
## ------------------------------------------------

frag <- lesson_fragment("sandpaper-fragment-with-child")
lsn <- Lesson$new(frag, jekyll = FALSE)
lsn$has_children # TRUE
lsn$episodes[[1]]$children # first episode shows 1 immediate child
lsn$trace_lineage(lsn$files[[1]]) # find recursive children of 1st episode
```

`lesson_fragment` *Example Lesson Fragments*

Description

Partial lessons mainly used for testing and demonstration purposes

Usage

```
lesson_fragment(name = "lesson-fragment")
```

Arguments

`name`	the name of the lesson fragment. Can be one of: • lesson-fragment • rmd-lesson • sandpaper-fragment • sandpaper-fragment with child

Value

a path to a lesson fragment whose contents are:

- `lesson-fragment` contains one `_episodes` directory with three files: "10-lunch.md", "14-looping-data-sets.md", and "17-scope.md"
- `rmd-fragment` contains one episode under `_episodes_rmd` called `01-test.Rmd`.
- `sandpaper-fragment` contains a trimmed-down Workbench lesson that has its R Markdown content pre-built
- `sandpaper-fragment-with-child` contains much of the same content as `sandpaper-fragment`, but the `episodes/index.Rmd` file references child documents.

Note

The lesson-fragment example was taken from the python novice gapminder lesson

Examples

```
lesson_fragment()
lesson_fragment("rmd-lesson")
lesson_fragment("sandpaper-fragment")
lesson_fragment("sandpaper-fragment-with-child")
```

`liquid_to_commonmark` *Convert liquid code blocks to commonmark code blocks*

Description

Liquid code blocks are generally codified by

Usage

```
liquid_to_commonmark(block, make_rmd = FALSE)
```

Arguments

`block`	a code block
`make_rmd`	if `TRUE`, the language will be wrapped in curly braces to be evaluated by RMarkdown

Details

`print("code goes " + "here")`

: .language-python

However, there is a simpler syntax that we can use:

```
print("code goes " + "here")
```

This will take in a code block and convert it so that it will no longer use the liquid tag (which we have added as a "ktag" attribute for "kramdown" tag)

Value

the node, invisibly.

Examples

```
frg1 <- Lesson$new(lesson_fragment())
frg2 <- frg1$clone(deep = TRUE)
py1  <- get_code(frg1$episodes[["17-scope.md"]]$body, ".language")
py2  <- get_code(frg2$episodes[["17-scope.md"]]$body, ".language")
py1
invisible(lapply(py1, liquid_to_commonmark, make_rmd = FALSE))
invisible(lapply(py2, liquid_to_commonmark, make_rmd = TRUE))
py1
py2
```

`make_div_table` *Create a table of divs in an episode*

Description

Create a table of divs in an episode

Usage

```
make_div_table(yrn)
```

Arguments

yrn	a tinkr::yarn or Episode object.

Value

a data frame with the following columns:

- path: path to the file, relative to the lesson
- div: the type of div
- pb_label: the label of the div
- line: the line number of the div label

`make_pandoc_alt` *Add alt text to images when transforming from jekyll to sandpaper*

Description

Add alt text to images when transforming from jekyll to sandpaper

Usage

```
make_pandoc_alt(images)
```

Arguments

`images`	a xml_nodeset of image nodes

Value

the images, invisibly with a new alt attribute and text removed

`set_alt_attr` *Set the alt text for a nodeset of images*

Description

This finds the attribute curly braces after an image declaration, extracts the alt text, and adds it as an attribute to the image, which is useful in parsing the XML, and will not affect rendering.

Usage

```
set_alt_attr(images, xpath, ns)
```

Arguments

`images`	a nodeset of images
`xpath`	an XPath expression that finds the first curly brace immediately after a node.
`ns`	the namespace of the XML

Value

the nodeset, invisibly.

Note

this function assumes that the images entering have a curly brace following.

`throw_heading_warnings` *Throw a validation report as a single message*

Description

Collapse a variable number of validation reports into a single message that can be formatted for the CLI or GitHub.

Usage

```
throw_heading_warnings(VAL)

throw_div_warnings(VAL)

throw_link_warnings(VAL)
```

Arguments

`VAL`	`[data.frame]` a validation report derived from one of the `validate` functions.

Details

One of the key features of pegboard is the ability to parse and validate markdown elements. These functions provide a standard way of creating the reports that are for the user based on whether or not they are on the CLI or on GitHub. The prerequisites of these functions are the input data frame (generated from the actual validation function) and an internal set of known templating vectors that contain templates for each test to show the actual error along with general information that can help correct the error (see below).

Input Data Frame:

The validations are initially reported in a data frame that has the following properties:

- one row per element
- columns that indicate the parsed attributes of the element, source file, source position, and the actual element XML node object.
- boolean columns that indicate the tests for each element, used with `collect_labels()` to add a "labels" column to the data.

Templating vectors:

These vectors come in two forms `[thing]_tests` and `[thing]_info` (e.g. for `validate_links()`, we have `link_tests` and `link_info`). These are named vectors that match the boolean columns of the data frame produced by the validation function. The `[thing]_tests` vector contains templates that describes the error and shows the text that caused the error. The `[thing]_info` contains general information about how to address that particular error. For example, one common link error is that a link is not descriptive (e.g. the link text says "click here"). The column in the `VAL` data frame that contains the result of this test is called "descriptive", so if we look at the values from the link info and tests vectors:

```
link_info["descriptive"]
#>                                                                  descriptive
#> "Avoid uninformative link phrases <https://webaim.org/techniques/hypertext/link_text#uninformati
link_tests["descriptive"]
#>                                  descriptive
#> "[uninformative link text]: [{text}]({orig})"
```

If the `throw_*_warnings()` functions detect any errors, they will use the info and tests vectors to construct a composite message.

Process:

The `throw_*_warnings()` functions all do the same basic procedure (and indeed could be consolidated into a single function in the future)

1. pass data to `collect_labels()`, which will parse the `[thing]_tests` templating vector and label each failing element in `VAL` with the appropriate failure message
2. gather the source information for each failure
3. pass failures with the `[thing]_info` elements that matched the unique failures to `issue_warning()`

Value

NULL, invisibly. This is used for it's side-effect of formatting and issuing messages via `issue_warning()`.

See Also

`validate_links()`, `validate_divs()`, and `validate_headings()` for input sources for these functions.

`trim_fence` *Trim div fences from output*

Description

Trim div fences from output

Usage

```
trim_fence(nodes)
```

Arguments

`nodes`	an xml_nodeset whose first and last node are div fences

Value

the nodeset without div fences

validate_divs *Validate Callout Blocks for sandpaper episodes*

Description

The Carpentries Workbench uses pandoc fenced divs to create special blocks within the lesson for learners and instructors to provide breaks in the narrative flow for focus on specific tasks or caveats. These fenced divs look something like this:

Usage

```
validate_divs(yrn)

div_is_known(div_table)

KNOWN_DIVS

div_tests

div_info
```

Arguments

`yrn`	a tinkr::yarn or Episode object.
`div_table`	a data frame derived from `make_div_table()`

Format

An object of class `character` of length 13.

An object of class `character` of length 1.

An object of class `character` of length 1.

Details

```
::: callout

### Hello!

This is a callout block

:::
```

Lessons created with The Carpentries Workbench are expected to have the following fenced divs:

- objectives (top)
- questions (top)
- keypoints (bottom)

The following fenced divs can occur in the lesson, but are not required:

- prereq
- callout
- challenge
- solution (nested inside challenge)
- hint (nested inside challenge)
- discussion
- checklist
- testimonial
- tab (can only contain text, images, and code blocks)

Any other div names will produce structure in the resulting DOM, but they will not have any special visual styling.

Value

a data frame with the following columns:

- div: the type of div
- label: the label of the div
- line: the line number of the div label
- is_known: a logical value if the div is a known type (`TRUE`) or not (`FALSE`)

Index