Package 'rsyntax'

July 23, 2025

Type Package

Title Extract Semantic Relations from Text by Querying and Reshaping Syntax

Version 0.1.4

Date 2022-06-06

Author Kasper Welbers and Wouter van Atteveldt

Maintainer Kasper Welbers <kasperwelbers@gmail.com>

Depends R (>= 3.2.0)

Imports igraph, tidyselect, methods, stringi, digest, rlang, magrittr, tokenbrowser, base64enc, png, data.table (>= 1.11.8)

Enhances spacyr

LazyData true

Encoding UTF-8

Description Various functions for querying and reshaping dependency trees, as for instance created with the 'spacyr' or 'udpipe' packages. This enables the automatic extraction of useful semantic relations from texts, such as quotes (who said what) and clauses (who did what). Method proposed in Van Atteveldt et al. (2017) <doi:10.1017/pan.2016.12>.

License GPL-3

RoxygenNote 7.1.2

Suggests testthat

NeedsCompilation no

Repository CRAN

Date/Publication 2022-06-07 04:30:05 UTC

Contents

add_span_quotes *Add span quotes to a source-quote annotations*

Description

Quotes can span across sentences, which makes it impossible to find them based on dependency tree quories. This function can be used as post-processing, AFTER using tqueries to find 'source' and 'quote' nodes, to add some of these quotes.

The quotes themselves are often easy to detect due to the use of quotation marks. There are two common ways of indicating the sources.

Firstly, the source might be used before the start of the quote (Steve said: "hey a quote!". "I like quotes!"). Secondly, the source might be implied in the sentence where the quotes starts, or the sentence before that (Steve was mad. "What a stupid way of quoting me!").

In the first case, the source can be found with a tquery. If there is a source (source_val) in the quote_col that is linked to a part of the quote (quote_val), this function will add the rest of the quote.

In the second case, we can look for candidates near the beginning of the quote. The candidate criteria can be specified as tqueries

Usage

```
add_span_quotes(
  tokens,
  text_col,
  quote_col = "quotes",
  source_val = "source",
  quote_val = "quote",
  tqueries = NULL,
  par_col = NULL,
  space_col = NULL,
  lag_sentences = 1,
  add_quote_symbols = NULL,
  quote_subset = NULL,
  copy = TRUE
)
```

Arguments

tokens	A tokenIndex with rsyntax annotations for 'sources' and 'quotes'
text_col	The column with the text (often 'token' or 'word')
quote_col	The column that contains the quote annotations
source_val	The value in quote_col that indicates the source
quote_val	The value in quote_col that indicates the quote

tqueries — A list of tqueries, that will be performed to find source candidates. The order of the queries determines which source candidates are preferred. It would make sense to use the same value as in source_val in the 'label' argument for the tquery.

par_col — If available in the parser output, the column with the paragraph id. We can assume that quotes do not span across paragraphs. By using this argument, quotes that are not properly closed (uneven number of quotes) will stop at the end of the paragraph

space_col — If par_col is not used, paragraphs will be identified based on hard enters in the text_col. In some parsers, there is an additional "space" column that hold the whitespace and linebreaks, which can be included here.

lag_sentences — The max number of sentences looked backwards to find source candidates. Default is 1, which means the source candidates have to occur in the sentence where the quote begins (lag = 0) or the sentence before that (lag = 1)

add_quote_symbols — Optionally, add additional punctuation symbols for finding quotation marks. In some contexts and languages it makes sense to add single quotes, but in that case it is oftne necessary to also use the quote_subset argument. For instance, in Spacy (and probably other UD based annotations), single quotes in posessives (e.g., Bob's, scholars') have a PART POS tag, whereas quotation symbols have PUNCT, NOUN, VERB, or ADJ (for some reason).

quote_subset — Optionally, an expression to be evaluated on the columns of 'tokens' for selecting/deselecting tokens that can/cant be quotation marks. For example, pos != "PART" can be used for the example mentioned in add_quote_symbols.

copy — If TRUE, deep copy the data.table (use if output tokens do not overwrite input tokens)

Value

the tokenIndex

Examples

```
## This function is best used after first annotating regular quotes
## Here we first apply 3 tqueries for annotating quotes in spacy tokens



tokens = tokens_spacy[tokens_spacy$doc_id == 'text6',]

verbs = c("tell", "show", "acknowledge", "admit", "affirm", "allege",
  "announce", "assert", "attest", "avow", "call", "claim", "comment",
  "concede", "confirm", "declare", "deny", "exclaim", "insist", "mention",
  "note", "post","predict", "proclaim", "promise", "reply", "remark",
  "report", "say", "speak", "state", "suggest", "talk", "tell", "think",
  "warn","write", "add")

direct = tquery(lemma = verbs, label='verb',
   children(req=FALSE, relation = c('npadvmod'), block=TRUE),
```

```
   children(relation=c('su','nsubj','agent','nmod:agent'), label='source'),
   children(label='quote'))

nosrc = tquery(pos='VERB*',
   children(relation= c('su', 'nsubj', 'agent', 'nmod:agent'), label='source'),
   children(lemma = verbs, relation='xcomp', label='verb',
    children(relation=c("ccomp","dep","parataxis","dobj","nsubjpass","advcl"), label='quote')))

according = tquery(label='quote',
   children(relation='nmod:according_to', label='source',
        children(label='verb')))

tokens = annotate_tqueries(tokens, 'quote', dir=direct, nos=nosrc, acc=according)
tokens

## now we add the span quotes. If a span quote is found, the algorithm will first
## look for already annotated sources as source candidates. If there are none,
## additional tqueries can be used to find candidates. Here we simply look for
## the most recent PERSON entity

tokens = tokens_spacy[tokens_spacy$doc_id == 'text6',]
tokens = annotate_tqueries(tokens, 'quote', dir=direct, nos=nosrc, acc=according)


last_person = tquery(entity = 'PERSON*', label='source')
tokens = add_span_quotes(tokens, 'token',
                         quote_col = 'quote', source_val = 'source', quote_val = 'quote',
                          tqueries=last_person)
tokens

## view as full text
syntax_reader(tokens, annotation = 'quote', value = 'source')
```

AND *Use AND search in tquery*

Description

Use AND search in tquery

Usage

```
AND(...)
```

Arguments

...	name-value pairs for look-up terms. see ?query.

Value

A list, to be used as input to tquery

Examples

```
tquery(AND(lemma = 'walk', POS='Noun'))   ## is also the default
```

`annotate` *Annotate a tokenlist based on rsyntax queries*

Description

This function has been renamed to annotate_tqueries.

Usage

```
annotate(
  tokens,
  column,
  ...,
  block = NULL,
  fill = TRUE,
  overwrite = FALSE,
  block_fill = FALSE,
  copy = TRUE,
  verbose = FALSE
)
```

Arguments

`tokens`	A tokenIndex data.table, or any data.frame coercible with as_tokenindex.
`column`	The name of the column in which the annotations are added. The unique ids are added as column_id
`...`	One or multiple tqueries, or a list of queries, as created with tquery. Queries can be given a named by using a named argument, which will be used in the annotation_id to keep track of which query was used.
`block`	Optionally, specify ids (doc_id - sentence - token_id triples) that are blocked from querying and filling (ignoring the id and recursive searches through the id).
`fill`	Logical. If TRUE (default) also assign the fill nodes (as specified in the tquery). Otherwise these are ignored
`overwrite`	If TRUE, existing column will be overwritten. Otherwise (default), the exsting annotations in the column will be blocked, and new annotations will be added. This is identical to using multiple queries.

block_fill	If TRUE (and overwrite is FALSE), the existing fill nodes will also be blocked. In other words, the new annotations will only be added if the
copy	If TRUE (default), the data.table is copied. Otherwise, it is changed by reference. Changing by reference is faster and more memory efficient, but is not predictable R style, so is optional.
verbose	If TRUE, report progress (only usefull if multiple queries are given)

Details

Apply queries to extract syntax patterns, and add the results as two columns to a tokenlist. One column contains the ids for each hit. The other column contains the annotations. Only nodes that are given a name in the tquery (using the 'label' parameter) will be added as annotation.

Note that while queries only find 1 node for each labeld component of a pattern (e.g., quote queries have 1 node for "source" and 1 node for "quote"), all children of these nodes can be annotated by settting fill to TRUE. If a child has multiple ancestors, only the most direct ancestors are used (see documentation for the fill argument).

Value

The tokenIndex with the annotation columns

Examples

```
## spacy tokens for: Mary loves John, and Mary was loved by John
tokens = tokens_spacy[tokens_spacy$doc_id == 'text3',]

## two simple example tqueries
passive = tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("agent"), label = "subject"))
active =  tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("nsubj", "nsubjpass"), label = "subject"))


tokens = annotate_tqueries(tokens, "clause", pas=passive, act=active)
tokens
if (interactive()) plot_tree(tokens, annotation='clause')
```

---

annotate_nodes *Annotate a tokenlist based on rsyntaxNodes*

Description

Use rsyntaxNodes, as created with tquery and apply_queries, to annotate a tokenlist. Three columns will be added: a unique id for the query match, the labels assigned in the tquery, and a column with the fill level (0 is direct match, 1 is child of match, 2 is grandchild, etc.).

Usage

```
annotate_nodes(tokens, nodes, column)
```

Arguments

tokens	A tokenIndex data.table, or any data.frame coercible with as_tokenindex.
nodes	An rsyntaxNodes A data.table, as created with apply_queries. Can be a list of multiple data.tables.
column	The name of the column in which the annotations are added. The unique ids are added as [column]_id, and the fill values are added as [column]_fill.

Details

Note that you can also directly use annotate.

Value

The tokenIndex data.table with the annotation columns added

Examples

```
## spacy tokens for: Mary loves John, and Mary was loved by John
tokens = tokens_spacy[tokens_spacy$doc_id == 'text3',]

## two simple example tqueries
passive = tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("agent"), label = "subject"))
active =  tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("nsubj", "nsubjpass"), label = "subject"))

nodes = apply_queries(tokens, pas=passive, act=active)
annotate_nodes(tokens, nodes, 'clause')
```

---

annotate_tqueries *Annotate a tokenlist based on rsyntax queries*

Description

Apply queries to extract syntax patterns, and add the results as three columns to a tokenlist. The first column contains the ids for each hit. The second column contains the annotation label. The third column contains the fill level (which you probably won't use, but is important for some functionalities). Only nodes that are given a name in the tquery (using the 'label' parameter) will be added as annotation.

Note that while queries only find 1 node for each labeld component of a pattern (e.g., quote queries have 1 node for "source" and 1 node for "quote"), all children of these nodes can be annotated by settting fill to TRUE. If a child has multiple ancestors, only the most direct ancestors are used (see documentation for the fill argument).

Usage

```
annotate_tqueries(
  tokens,
  column,
  ...,
  block = NULL,
  fill = TRUE,
  overwrite = NA,
  block_fill = FALSE,
  copy = TRUE,
  verbose = FALSE
)
```

Arguments

tokens	A tokenIndex data.table, or any data.frame coercible with as_tokenindex.
column	The name of the column in which the annotations are added. The unique ids are added as column_id
...	One or multiple tqueries, or a list of queries, as created with tquery. Queries can be given a named by using a named argument, which will be used in the annotation_id to keep track of which query was used.
block	Optionally, specify ids (doc_id - sentence - token_id triples) that are blocked from querying and filling (ignoring the id and recursive searches through the id).
fill	Logical. If TRUE (default) also assign the fill nodes (as specified in the tquery). Otherwise these are ignored
overwrite	Applies if column already exists. If TRUE, existing column will be overwritten. If FALSE, the existing annotations in the column will be blocked, and new annotations will be added. This is identical to using multiple queries.
block_fill	If TRUE (and overwrite is FALSE), the existing fill nodes will also be blocked. In other words, the new annotations will only be added if the
copy	If TRUE (default), the data.table is copied. Otherwise, it is changed by reference. Changing by reference is faster and more memory efficient, but is not predictable R style, so is optional.
verbose	If TRUE, report progress (only usefull if multiple queries are given)

Value

The tokenIndex data.table with the annotation columns added

Examples

```
## spacy tokens for: Mary loves John, and Mary was loved by John
tokens = tokens_spacy[tokens_spacy$doc_id == 'text3',]

## two simple example tqueries
passive = tquery(pos = "VERB*", label = "predicate",
```

```
                children(relation = c("agent"), label = "subject"))
active =  tquery(pos = "VERB*", label = "predicate",
                children(relation = c("nsubj", "nsubjpass"), label = "subject"))

tokens = annotate_tqueries(tokens, "clause", pas=passive, act=active)
tokens

if (interactive()) plot_tree(tokens, annotation='clause')
```

---

apply_queries *Apply queries created with tquery*

Description

Apply queries created with tquery

Usage

```
apply_queries(
  tokens,
  ...,
  as_chain = FALSE,
  block = NULL,
  check = FALSE,
  fill = TRUE,
  return_wide = FALSE,
  verbose = FALSE
)
```

Arguments

tokens	A tokenIndex data.table, or any data.frame coercible with as_tokenindex.
...	tqueries, as created with tquery. Can also be a list with tquery functions. It is recommended to use named arguments/lists, to name the tqueries.
as_chain	If TRUE, Nodes that have already been assigned assigned earlier in the chain will be ignored (see 'block' argument).
block	Optionally, specify ids (doc_id - sentence - token_id triples) where find_nodes will stop (ignoring the id and recursive searches through the id). Can also be a data.table returned by (a previous) apply_queries, in which case all ids are blocked.
check	If TRUE, return a warning if nodes occur in multiple patterns, which could indicate that the find_nodes query is not specific enough.
fill	If TRUE (default) the fill nodes are added. Otherwise these are ignored, even if the queries include fill()
return_wide	If TRUE, return nodes in wide format.
verbose	If TRUE, report progress (only useful if multiple queries are used)

Value

A data.table in which each row is a node for which all conditions are satisfied, and each column is one of the linked nodes (parents / children) with names as specified in the label argument.

Examples

```
## spacy tokens for: Mary loves John, and Mary was loved by John
tokens = tokens_spacy[tokens_spacy$doc_id == 'text3',]

## two simple example tqueries
passive = tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("agent"), label = "subject"))
active =  tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("nsubj", "nsubjpass"), label = "subject"))

nodes = apply_queries(tokens, pas=passive, act=active)
nodes
```

---

as_tokenindex *Prepare a tokenIndex*

Description

Creates a tokenIndex data.table. Accepts any data.frame given that the required columns (doc_id, sentence, token_id, parent, relation) are present. The names of these columns must be one of the values specified in the respective arguments.

The data in the data.frame will not be changed, with three exceptions. First, the columnnames will be changed if the default values are not used. Second, if a token has itself as its parent (which in some parsers is used to indicate the root), the parent is set to NA (as used in other parsers) to prevent infinite cycles. Third, the data will be sorted by doc_id, sentence, token_id.

Usage

```
as_tokenindex(
  tokens,
  doc_id = c("doc_id", "document_id"),
  sentence = c("sentence", "sentence_id"),
  token_id = c("token_id"),
  parent = c("parent", "head_token_id"),
  relation = c("relation", "dep_rel"),
  paragraph = NULL
)
```

Arguments

tokens	A data.frame, data.table, or tokenindex.
doc_id	candidate names for the document id columns

sentence	candidate names for sentence (id/index) column
token_id	candidate names for the token id column. Has to be numeric (Some parsers return token_id's as numbers with a prefix (t_1, w_1))
parent	candidate names for the parent id column. Has to be numeric
relation	candidate names for the relation column
paragraph	Optionally, the name of a column with paragraph ids. This is only necessary if sentences are numbered per paragraph, and therefore not unique within documents. If given, sentences are re-indexed to be unique within documents.

Value

a tokenIndex

Examples

```
as_tokenindex(tokens_corenlp)
```

BREAK *A special NOT condition if depth > 1*

Description

If depth > 1 in the children, parents or fill function, the children/parents will be retrieved recursively (i.e. children, children of children, etc.). If the look-up conditions (e.g., relation = 'nsubj') are not satisfied, a node will not be matched by the query, but the search will still continue for it's parents/children. The special BREAK look-up function allows you to specify a condition for breaking the recursive loop (lending it's name from the 'break' in a for loop). An example is that you might want to stop the recursive loop in a custom_fill() once it encounters a nested sentence, such as a relative clause: custom_fill(BREAK(relation = 'relcl')).

Usage

```
BREAK(...)
```

Arguments

...	name-value pairs for look-up terms. see ?query.

Value

A list, to be used as input to tquery

Examples

```
tquery(NOT(POS='Noun'))
```

cast_text *Cast annotations to text*

Description

Cast labeled tokens to sentences.

Usage

```
cast_text(tokens, annotation, ..., text_col = "token", na.rm = T)
```

Arguments

tokens	A tokenIndex
annotation	The name of annotations (the "column" argument in annotate_tqueries)
...	Optionally, group annotations together. Named arguments can be given where the name is the new group, and the value is a character vector with values in the annotation column. For example, text = c('verb','predicate') would group the 'verb' and 'predicate' nodes together under the name 'text'.
text_col	The name of the column in tokens with the text. Usually this is "token", but some parsers use alternatives such as 'word'.
na.rm	If true (default), drop tokens where annotation id is NA (i.e. tokens without labels)

Value

a data.table

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text3',]

## two simple example tqueries
passive = tquery(pos = "VERB*", label = "verb", fill=FALSE,
                 children(relation = "agent",
                          children(label="subject")),
                 children(relation = "nsubjpass", label="object"))
active =  tquery(pos = "VERB*", label = "verb", fill=FALSE,
                 children(relation = c("nsubj", "nsubjpass"), label = "subject"),
                 children(relation = "dobj", label="object"))

tokens = annotate_tqueries(tokens, "clause", pas=passive, act=active, overwrite=T)

cast_text(tokens, 'clause')

## group annotations
cast_text(tokens, 'clause', text = c('verb','object'))
```

```
## use grouping to sort
cast_text(tokens, 'clause', subject = 'subject',
                            verb = 'verb', object = 'object')
```

---

chop *Chop of a branch of the tree*

Description

Using the query language for tquery, chop of the branch down from the node that is found

Usage

```
chop(.tokens, ...)
```

Arguments

.tokens	A tokenIndex
...	Arguments passed to tquery. For instance, relation = 'punct' cuts off all punctuation dependencies (in universal dependencies)

Value

A tokenIndex with the rows of the nodes in the selected branches removed

Examples

```
spacy_conjunctions <- function(tokens) {
  no_fill = c('compound*','case', 'relcl')
  tq = tquery(label='target', NOT(relation = 'conj'),
              rsyntax::fill(NOT(relation = no_fill), max_window = c(Inf,0)),
              children(relation = 'conj', label='origin',
                       rsyntax::fill(NOT(relation = no_fill), max_window=c(0,Inf))))
  tokens = climb_tree(tokens, tq)
  chop(tokens, relation = 'cc')
}

## spacy tokens for "Bob and John ate bread and drank wine"
tokens = tokens_spacy[tokens_spacy$doc_id == 'text5',]

tokens = spacy_conjunctions(tokens)
tokens

if (interactive()) plot_tree(tokens)
```

climb_tree *Have a node adopt its parent's position*

Description

given a tquery that identfies a node labeled "origin", that has a parent labeled "target", recursively have child adopt the parent's position (parent and relation column) and adopt parents fill nodes. only_new restricts adding fill nodes to relations that child does not already have. This seems to be a good heuristic for dealing with argument drop

Usage

```
climb_tree(
  .tokens,
  tq,
  unpack = TRUE,
  isolate = TRUE,
  take_fill = TRUE,
  give_fill = TRUE,
  only_new = "relation",
  max_iter = 200
)
```

Arguments

.tokens	A tokenIndex
tq	A tquery. Needs to have a node labeled "origin" that has a parent labeled "target"
unpack	If TRUE (default), create separate branches for the parent and the node that inherits the parent position
isolate	If unpack is TRUE and isolate is TRUE (default is FALSE), isolate the new branch by recursively unpacking
take_fill	If TRUE (default), give the node that will inherit the parent position a copy of the parent children (but only if it does not already have children with this relation; see only_new)
give_fill	If TRUE (default), copy the children of the node that will inherit the parent position to the parent (but only if it does not already have children with this relation; see only_new)
only_new	A characetr vector giving one or multiple column names that need to be unique for take_fill and give_fill
max_iter	The climb tree function repeatedly resolves the first conjunction it encounters in a sentence. This can lead to many iterations for sentences with many (nested) conjunctions. It could be the case that in unforseen cases or with certain parsers an infinite loop is reached, which is why we use a max_iter argument that breaks the loop and sends a warning if the max is reached.

Value

The reshaped tokenIndex

Examples

```
spacy_conjunctions <- function(tokens) {
  no_fill = c('compound*','case', 'relcl')
  tq = tquery(label='target', NOT(relation = 'conj'),
              rsyntax::fill(NOT(relation = no_fill), max_window = c(Inf,0)),
              children(relation = 'conj', label='origin',
                       rsyntax::fill(NOT(relation = no_fill), max_window=c(0,Inf))))
  tokens = climb_tree(tokens, tq)
  chop(tokens, relation = 'cc')
}

## spacy tokens for "Bob and John ate bread and drank wine"
tokens = tokens_spacy[tokens_spacy$doc_id == 'text5',]

tokens = spacy_conjunctions(tokens)

tokens

if (interactive()) plot_tree(tokens)
```

---

copy_fill *Copy nodes*

Description

Copy nodes

Usage

```
copy_fill(
  .tokens,
  from_node,
  to_node,
  subset = NULL,
  subset_fill = NULL,
  only_new = NULL
)
```

Arguments

.tokens	A tokenIndex in which nodes are selected with select_nodes.
from_node	The name of the node from which fill is copied
to_node	The name of the node to which fill is copied

subset	A subset expression (that evaluates to a logical vector). The token column for each labeled node in the tquery can be referred to as label$column.
subset_fill	A subset on the fill nodes. Can only directly use token column. For example, use pos == 'VERB' to copy only verbs
only_new	If TRUE, direct fill children will only be copied to to_node if it does not already have nodes of this relation. This is a good heuristic for dealing with argument drop.

Value

A tokenIndex with a .nodes attribute

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text1',]

tq = tquery(label='object', relation='dobj')

tokens2 = select_nodes(tokens, tq)
selected_nodes(tokens2)

tokens3 = copy_nodes(tokens2, 'object', 'new_object')
copy_fill(tokens3, 'object', 'new_object')
```

---

copy_nodes *Copy nodes*

Description

Copy nodes

Usage

```
copy_nodes(
  .tokens,
  node,
  new,
  subset = NULL,
  keep_relation = TRUE,
  copy_fill = FALSE,
  subset_fill = NULL,
  only_new = NULL
)
```

Arguments

.tokens	A tokenIndex in which nodes are selected with select_nodes.
node	The name of the node that is to be copied
new	The name given to the copy
subset	A subset expression (that evaluates to a logical vector). The token column for each labeled node in the tquery can be referred to as label$column.
keep_relation	If FALSE, remove relation (making node a root)
copy_fill	If TRUE, also copy the fill
subset_fill	A subset on the fill nodes. Can only directly use token column. For example, use pos == 'VERB' to copy only verbs
only_new	If TRUE, direct fill children will only be copied to to_node if it does not already have nodes of this relation. This is a good heuristic for dealing with argument drop.

Value

A tokenIndex with a .nodes attribute

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text1',]

tq = tquery(label='object', relation='dobj')

tokens2 = select_nodes(tokens, tq)
selected_nodes(tokens2)

copy_nodes(tokens2, 'object', 'new_object')

tokens3 = copy_nodes(tokens2, 'object', 'new_object', copy_fill=TRUE)


if (interactive()) plot_tree(tokens3, token, pos)
```

---

custom_fill *Specify custom fill behavior*

Description

If a tquery(), parents() or children() function has set a label, all children of the matched node (that are not matched by another query) will also be given this label. This is called the 'fill' heuristic. The custom_fill() function can be used to give more specific conditions for which children need to be labeled.

The function can be used almost identically to the children() function. The specification of the look-up conditions works in the same way. NOTE that custom_fill, just like the children() function,

should be passed as an unnamed argument, and NOT to the 'fill' argument (which is the boolean argument for whether fill should be used)

For the custom_fill function, the special BREAK() look-up function is particularly powerful. custom_fill will recursively search for children, children of children, etc. The look-up conditions in custom_fill determine which of all these direct and indirect children to label. Often, however, you would want to the recursive loop to 'break' when certain conditions are met. For instance, to ignore children in a relative clause: custom_fill(BREAK(relation = 'relcl'))

Usage

```
custom_fill(
  ...,
  g_id = NULL,
  depth = Inf,
  connected = FALSE,
  max_window = c(Inf, Inf),
  min_window = c(0, 0)
)
```

Arguments

`...`	Accepts two types of arguments: name-value pairs for finding nodes (i.e. rows), and functions to look for parents/children of these nodes. The name in the name-value pairs need to match a column in the data.table, and the value needs to be a vector of the same data type as the column. By default, search uses case sensitive matching, with the option of using common wildcards (* for any number of characters, and ? for a single character). Alternatively, flags can be used to to change this behavior to 'fixed' (__F), 'igoring case' (__I) or 'regex' (__R). See details for more information. If multiple name-value pairs are given, they are considered as AND statements, but see details for syntax on using OR statements, and combinations. To look for parents and children of the nodes that are found, you can use the parents and children functions as (named or unnamed) arguments. These functions have the same query arguments as tquery, but with some additional arguments.
`g_id`	Find nodes by global id, which is the combination of the doc_id, sentence and token_id. Passed as a data.frame or data.table with 3 columns: (1) doc_id, (2) sentence and (3) token_id.
`depth`	A positive integer, determining how deep parents/children are sought. 1 means that only direct parents and children of the node are retrieved. 2 means children and grandchildren, etc. All parents/children must meet the filtering conditions (... or g_id)
`connected`	Controls behavior if depth > 1 and filters are used. If FALSE, all parents/children to the given depth are retrieved, and then filtered. This way, grandchildren that satisfy the filter conditions are retrieved even if their parents do not satisfy the conditions. If TRUE, the filter is applied at each level of depth, so that only fully connected branches of nodes that satisfy the conditions are retrieved.

max_window	Set the max token distance of the children/parents to the node. Has to be either a numerical vector of length 1 for distance in both directions, or a vector of length 2, where the first value is the max distance to the left, and the second value the max distance to the right. Default is c(Inf, Inf) meaning that no max distance is used.
min_window	Like max_window, but for the min distance. Default is c(0,0) meaning that no min is used.

Value

Should not be used outside of tquery

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text4',]

## custom fill rule that ignores relative clauses
no_relcl_fill = custom_fill(BREAK(relation='relcl'))

## add custom fill as argument in children(). NOTE that it should be
## passed as an unnamed argument (and not to the fill boolean argument)
tq = tquery(label = 'verb', pos='VERB', fill=FALSE,
         children(label = 'subject', relation = 'nsubj', no_relcl_fill),
         children(label = 'object', relation = 'dobj', no_relcl_fill))

tokens = annotate_tqueries(tokens, "clause", tq)
tokens
```

---

dutch *Dutch lemma*

Description

Various categories of lemma, for use in syntax queries

Usage

```
data(dutch)
```

Format

list

fill *Specify custom fill behavior*

Description

This is soft deprecated, with the new preferred function being custom_fill to avoid namespace conflicts with tidyr::fill() and data.table::fill()

Usage

```
fill(...)
```

Arguments

...	passes to custom_fill

Value

Should not be used outside of tquery

get_branch_id *Add the branch id as a column to the tokenindex*

Description

After splitting trees into branches

Usage

```
get_branch_id(tokens)
```

Arguments

tokens	A tokenindex

Value

the tokenindex

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text4',]
tokens = as_tokenindex(tokens)


tokens2 = isolate_branch(tokens, relation = 'relcl', copy_parent = TRUE)
get_branch_id(tokens2)
```

get_long_ids *Get ids in various forms to extract token_ids*

Description

Get ids in various forms to extract token_ids

Usage

```
get_long_ids(..., select = NULL, with_fill = FALSE)
```

Arguments

...	Either a data.table with the columns doc_id, sentence and token_id, or the output of apply_queries
select	If not null, a character vector for selecting column names
with_fill	If TRUE, include the ids of the fill nodes

Value

A data.table with the columns doc_id, sentence and token_id

get_nodes *Transform the nodes to long format and match with token data*

Description

Transform the nodes to long format and match with token data

Usage

```
get_nodes(tokens, nodes, use = NULL, token_cols = c("token"))
```

Arguments

tokens	A tokenIndex data.table, or any data.frame coercible with as_tokenindex.
nodes	A data.table, as created with apply_queries. Can be a list of multiple data.tables.
use	Optionally, specify which columns from nodes to add. Other than convenient, this is slighly different from subsetting the columns in 'nodes' beforehand if fill is TRUE. When the children are collected, the ids from the not-used columns are still blocked (see 'block')
token_cols	A character vector, specifying which columns from tokens to include in the output

Value

A data.table with the nodes in long format, and the specified token_cols attached

Examples

```
## spacy tokens for: Mary loves John, and Mary was loved by John
tokens = tokens_spacy[tokens_spacy$doc_id == 'text3',]

## two simple example tqueries
passive = tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("agent"), label = "subject"))
active =  tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("nsubj", "nsubjpass"), label = "subject"))

nodes = apply_queries(tokens, pas=passive, act=active)
get_nodes(tokens, nodes)
```

---

isolate_branch *Isolate a branch in a dependency tree*

Description

cuts of a branch at the nodes that match the lookup arguents (...). A "tree_parent" column is added to the tokenindex, that indicates for the new roots which node the parent was.

Usage

```
isolate_branch(tokens, ..., copy_parent = TRUE, copy_parent_fill = TRUE)
```

Arguments

tokens	A tokenindex
...	lookup arguments to find the node to split. For example, isolate_branch(tokens, relation='relcl') isolates branches of which the top node (the new root) has the relation "relcl".
copy_parent	If TRUE (default) copy the parent of the branch and include it in the isolated branch
copy_parent_fill	If TRUE, also copy the parents fill nodes

Value

the tokenindex

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text4',]
tokens = as_tokenindex(tokens)

tokens2 = isolate_branch(tokens, relation = 'relcl', copy_parent = TRUE)
tokens2

if (interactive()) plot_tree(tokens2)
```

---

mutate_nodes *Mutate nodes*

Description

Mutate nodes

Usage

```
mutate_nodes(.tokens, node, ..., subset = NULL)
```

Arguments

.tokens	A tokenIndex in which nodes are selected with select_nodes.
node	The name of the node that is to be mutated
...	named arguments. The name should be a column in tokens
subset	A subset expression (that evaluates to a logical vector). The token column for each labeled node in the tquery can be referred to as label$column.

Value

A tokenIndex with a .nodes attribute

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text4',]

## use a tquery to label the nodes that you want to manipulate
tq = tquery(relation = "relcl", label = "relative_clause")

## apply query to select nodes
tokens2 = select_nodes(tokens, tq)

## as an example, we make the parent of the relative_clause
## nodes NA, effectively cutting of the relcl from the tree
tokens2 = mutate_nodes(tokens2, "relative_clause", parent=NA)

tokens2
```

nested_nodes *Search for parents or children in tquery*

Description

Enables searching for parents or children. Should only be used inside of the tquery function, or within other children/parents functions. Look-up conditions are specified in the same way as in the tquery function.

Multiple children() or parents() functions can be nested side by side. This works as an AND condition: the node must have all these parents/children (unless the req [required] argument is set to FALSE).

The custom_fill() function is used to include the children of a 'labeled' node. It can only be nested in a query if the label argument is not NULL, and by default will include all children of the node that have not been assigned to another node. If two nodes have a shared child, the child will be assigned to the closest node.

Usage

```
children(
  ...,
  g_id = NULL,
  label = NA,
  req = TRUE,
  depth = 1,
  connected = FALSE,
  fill = TRUE,
  block = FALSE,
  max_window = c(Inf, Inf),
  min_window = c(0, 0)
)

not_children(
  ...,
  g_id = NULL,
  depth = 1,
  connected = FALSE,
  max_window = c(Inf, Inf),
  min_window = c(0, 0)
)

parents(
  ...,
  g_id = NULL,
  label = NA,
  req = TRUE,
  depth = 1,
```

```
  connected = FALSE,
  fill = TRUE,
  block = FALSE,
  max_window = c(Inf, Inf),
  min_window = c(0, 0)
)

not_parents(
  ...,
  g_id = NULL,
  depth = 1,
  connected = FALSE,
  max_window = c(Inf, Inf),
  min_window = c(0, 0)
)
```

Arguments

...	Accepts two types of arguments: name-value pairs for finding nodes (i.e. rows), and functions to look for parents/children of these nodes. The name in the name-value pairs need to match a column in the data.table, and the value needs to be a vector of the same data type as the column. By default, search uses case sensitive matching, with the option of using common wildcards (* for any number of characters, and ? for a single character). Alternatively, flags can be used to to change this behavior to 'fixed' (__F), 'igoring case' (__I) or 'regex' (__R). See details for more information. If multiple name-value pairs are given, they are considered as AND statements, but see details for syntax on using OR statements, and combinations. To look for parents and children of the nodes that are found, you can use the parents and children functions as (named or unnamed) arguments. These functions have the same query arguments as tquery, but with some additional arguments.
g_id	Find nodes by global id, which is the combination of the doc_id, sentence and token_id. Passed as a data.frame or data.table with 3 columns: (1) doc_id, (2) sentence and (3) token_id.
label	A character vector, specifying the column name under which the selected tokens are returned. If NA, the column is not returned.
req	Can be set to false to not make a node 'required'. This can be used to include optional nodes in queries. For instance, in a query for finding subject - verb - object triples, make the object optional.
depth	A positive integer, determining how deep parents/children are sought. 1 means that only direct parents and children of the node are retrieved. 2 means children and grandchildren, etc. All parents/children must meet the filtering conditions (... or g_id)
connected	Controls behavior if depth > 1 and filters are used. If FALSE, all parents/children to the given depth are retrieved, and then filtered. This way, grandchildren that satisfy the filter conditions are retrieved even if their parents do not satisfy the

	conditions. If TRUE, the filter is applied at each level of depth, so that only fully connected branches of nodes that satisfy the conditions are retrieved.
`fill`	Logical. If TRUE (default), the default custom_fill() will be used. To more specifically control fill, you can nest the custom_fill function (a special version of the children function).
`block`	Logical. If TRUE, the node will be blocked from being assigned (labeld). This is mainly usefull if you have a node that you do not want to be assigned by fill, but also don't want to 'label' it. Essentially, block is shorthand for using label and then removing the node afterwards. If block is TRUE, label has to be NA.
`max_window`	Set the max token distance of the children/parents to the node. Has to be either a numerical vector of length 1 for distance in both directions, or a vector of length 2, where the first value is the max distance to the left, and the second value the max distance to the right. Default is c(Inf, Inf) meaning that no max distance is used.
`min_window`	Like max_window, but for the min distance. Default is c(0,0) meaning that no min is used.

Details

Having nested queries can be confusing, so we tried to develop the find_nodes function and the accompanying functions in a way that clearly shows the different levels. As shown in the examples, the idea is that each line is a node, and to look for parents or children, we put them on the next line with indentation (in RStudio, it should automatically align correctly when you press enter inside of the children() or parents() functions).

There are several flags that can be used to change search condition. To specify flags, add a double underscore and the flag character to the name in the name value pairs (...). By adding the suffix __R, query terms are considered to be regular expressions, and the suffix __I uses case insensitive search (for normal or regex search). If the suffix __F is used, only exact matches are valid (case sensitive, and no wildcards). Multiple flags can be combined, such as lemma__RI, or lemma__IR (order of flags is irrelevant)

The not_children and not_parents functions will make the matched children/parents a NOT condition. Note that this is different from using the NOT() look-up function. NOT operates at the node level, so you specify that a node should NOT be matched if certain conditions are met. the not_parents and not_children functions operate at the pattern level, so you can specify that a pattern is invalid if these parents/children are matched.

Next to the OR, AND, and NOT functions, children/parents functions can have the special BREAK function for cases where depth > 1. If depth > 1 in the children, parents or fill function, the children/parents will be retrieved recursively (i.e. children, children of children, etc.). If the look-up conditions (e.g., relation = 'nsubj') are not satisfied, a node will not be matched by the query, but the search will still continue for it's parents/children. The special BREAK look-up function allows you to specify a condition for breaking the recursive loop (lending it's name from the 'break' in a for loop). An example is that you might want to stop the recursive loop in a custom_fill() once it encounters a nested sentence, such as a relative clause: custom_fill(BREAK(relation = 'relcl')).

Value

Should not be used outside of tquery

NOT *Use NOT search in tquery*

Description

Use NOT search in tquery

Usage

```
NOT(...)
```

Arguments

...	name-value pairs for look-up terms. see ?query.

Value

A list, to be used as input to tquery

Examples

```
tquery(NOT(POS='Noun'))
```

OR *Use OR search in tquery*

Description

Use OR search in tquery

Usage

```
OR(...)
```

Arguments

...	name-value pairs for look-up terms. see ?query.

Value

A list, to be used as input to tquery

Examples

```
tquery(OR(lemma = 'walk', POS='Noun'))
```

plot_tree *Create an igraph tree from a sentence*

Description

Create an igraph tree from a token_index (as_tokenindex) or a data.frame that can be coerced to a tokenindex.

By default, all columns in the data are included as labels. This can be changes by using the ... argument.

Usage

```
plot_tree(
  tokens,
  ...,
  sentence_i = 1,
  doc_id = NULL,
  sentence = NULL,
  annotation = NULL,
  only_annotation = FALSE,
  pdf_file = NULL,
  allign_text = TRUE,
  ignore_rel = NULL,
  all_lower = FALSE,
  all_abbrev = NULL,
  textsize = 1,
  spacing = 1,
  use_color = TRUE,
  max_curve = 0.3,
  palette = grDevices::terrain.colors,
  rel_on_edge = F,
  pdf_viewer = FALSE,
  viewer_mode = TRUE,
  viewer_size = c(100, 100)
)
```

Arguments

tokens	A tokenIndex data.table, or any data.frame coercible with as_tokenindex. Can also be a corpustools tCorpus.
...	Optionally, select which columns to include as labels and how to present them. Can be quoted or unquoted names and expressions, using columns in the tokenIndex. For example, plot_tree(tokens, token, pos) will use the $token and $pos columns in tokens. You can also use expressions for easy controll of visulizations. For example: plot_tree(tokens, tolower(token), abbreviate(pos,1)). (note that abbreviate() is really usefull here)

sentence_i	By default, plot_tree uses the first sentence (sentence_i = 1) in the data. sentence_i can be changed to select other sentences by position (the i-th unique sentence in the data). Note that sentence_i does not refer to the values in the sentence column (for this use the sentence argument together with doc_id)
doc_id	Optionally, the document id can be specified. If so, sentence_i refers to the i-th sentence within the given document.
sentence	Optionally, the sentence id can be specified (note that sentence_i refers to the position). If sentence is given, doc_id has to be given as well.
annotation	Optionally, a column with an rsyntax annotation, to add boxes around the annotated nodes.
only_annotation	If annotation is given, only_annotation = TRUE will print only the nodes with annotations.
pdf_file	Directly save the plot as a pdf file
allign_text	If TRUE (default) allign text (the columns specified in ...) in a single horizontal line at the bottom, instead of following the different levels in the tree
ignore_rel	Optionally, a character vector with relation names that will not be shown in the tree
all_lower	If TRUE, make all text lowercase
all_abbrev	If an integer, abbreviate all text, with the number being the target number of characters.
textsize	A number to manually change the textsize. The function tries to set a suitable textsize for the plotting device, but if this goes wrong and now everything is broken and sad, you can multiply the textsize with the given number.
spacing	A number for scaling the distance between words (between 0 and infinity)
use_color	If true, use colors
max_curve	A number for controlling the allowed amount of curve in the edges.
palette	A function for creating a vector of n contiguous colors. See ?terrain.colors for standard functions and documentation
rel_on_edge	If TRUE, print relation label on edge instead of above the node
pdf_viewer	If TRUE, view the plot as a pdf. If no pdf_file is specified, the pdf will be saved to the temp folder
viewer_mode	By default, the plot is saved as a PNG embedded in a HTML and opened in the viewer. This hack makes it independent of the size of the plotting device and enables scrolling. By setting viewer_mode to False, the current plotting device is used.
viewer_size	A vector of length 2, that multiplies the width (first value) and height (second value) of the viewer_mode PNG

Value

plots a dependency tree.

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text3',]


if (interactive()) plot_tree(tokens, token, pos)

## plot with annotations
direct = tquery(label = 'verb', pos = 'VERB', fill=FALSE,
                children(label = 'subject', relation = 'nsubj'),
                children(label = 'object', relation = 'dobj'))
passive = tquery(label = 'verb', pos = 'VERB', fill=FALSE,
                 children(label = 'subject', relation = 'agent'),
                 children(label = 'object', relation = 'nsubjpass'))

if (interactive()) {
tokens %>%
   annotate_tqueries('clause', pas=passive, dir=direct) %>%
   plot_tree(token, pos, annotation='clause')
}
```

---

print.tQuery *S3 print for tQuery class*

Description

S3 print for tQuery class

Usage

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

Arguments

x	a tQuery
...	not used

Examples

```
q = tquery(label='quote',
           children(relation='nmod:according_to', label='source',
                    children(label='verb')))
q
```

quote_punctuation *Quote punctuation*

Description

Punctuation used in quotes, for use in syntax queries

Usage

```
data(quote_punctuation)
```

Format

character()

remove_fill *Remove fill*

Description

Like remove_nodes, but only removing the fill nodes

Usage

```
remove_fill(
  .tokens,
  node,
  rm_subset_fill = NULL,
  rm_subset = NULL,
  keep_shared = FALSE
)
```

Arguments

.tokens	A tokenIndex in which nodes are selected with select_nodes.
node	The name of the node that is to be mutated
rm_subset_fill	A subset on the fill nodes. Can only directly use token column. For example, use pos == 'VERB' to remove only verbs
rm_subset	A subset expression (that evaluates to a logical vector) to more specifically specify which nodes to remove. The token column for each labeled node in the tquery can be referred to as label$column.
keep_shared	If there is another node that has the same fill nodes, should the fill nodes that are shared also be removed?

Value

A tokenIndex with a .nodes attribute

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text1',]

## use a tquery to label the nodes that you want to manipulate
tq = tquery(pos = 'VERB',
            children(label = 'object', relation='dobj'))

## apply query to select nodes
tokens2 = select_nodes(tokens, tq)

remove_fill(tokens2, 'object')
```

`remove_nodes` *Remove nodes*

Description

Remove nodes

Usage

```
remove_nodes(
  .tokens,
  node,
  rm_subset = NULL,
  with_fill = TRUE,
  rm_subset_fill = NULL,
  keep_shared = FALSE
)
```

Arguments

`.tokens`	A tokenIndex in which nodes are selected with select_nodes.
`node`	The name of the node that is to be mutated
`rm_subset`	A subset expression (that evaluates to a logical vector) to more specifically specify which nodes to remove. The token column for each labeled node in the tquery can be referred to as label$column.
`with_fill`	If TRUE, also remove the fill nodes
`rm_subset_fill`	A subset on the fill nodes. Can only directly use token column. For example, use pos == 'VERB' to remove only verbs
`keep_shared`	If there is another node that has the same fill nodes, should the fill nodes that are shared also be removed?

Value

A tokenIndex with a .nodes attribute

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text1',]

## use a tquery to label the nodes that you want to manipulate
tq = tquery(pos = 'VERB',
            children(label = 'object', relation='dobj'))

## apply query to select nodes
tokens2 = select_nodes(tokens, tq)

remove_nodes(tokens2, 'object')
remove_nodes(tokens2, 'object', with_fill=FALSE)
```

---

reselect_nodes *Within a chain of reshape operations, reapply the tquery*

Description

Within a chain of reshape operations, reapply the tquery

Usage

```
reselect_nodes(.tokens)
```

Arguments

.tokens	A tokenIndex in which nodes are selected with select_nodes.

Value

A tokenIndex with a .nodes attribute

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text4',]

## use a tquery to label the nodes that you want to manipulate
tq = tquery(relation = "relcl", label = "relative_clause")

## apply query to select nodes
tokens2 = select_nodes(tokens, tq)

## reuses the tq, that is stored in tokens2
## this makes it easy to make the selection anew after a transformation
tokens2 = reselect_nodes(tokens2)
```

rsyntax_threads *Get the number of threads to be used by rsyntax functions*

Description

rsyntax relies heavily on the data.table package, which supports multithreading. By default, the number of threads set by data.table are used, as you can see with `getDTthreads`. With `set_rsyntax_threads` you can set the number of threads for rsyntax functions, without affecting the data.table settings.

Usage

```
rsyntax_threads()
```

Value

the setting for the number of threads used by rsyntax

Examples

```
rsyntax_threads()
```

selected_nodes *If select_nodes() is used, the selected nodes can be extracted with selected_nodes(). This is mainly for internal use, but it can also be usefull for debugging, and to controll loops of reshape operation (e.g. break if no selected nodes left)*

Description

If select_nodes() is used, the selected nodes can be extracted with selected_nodes(). This is mainly for internal use, but it can also be usefull for debugging, and to controll loops of reshape operation (e.g. break if no selected nodes left)

Usage

```
selected_nodes(.tokens)
```

Arguments

`.tokens`	A tokenIndex in which nodes are selected with select_nodes.

Value

A tokenIndex with a .nodes attribute

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text4',]

## use a tquery to label the nodes that you want to manipulate
tq = tquery(relation = "relcl", label = "relative_clause")

## apply query to select nodes
tokens2 = select_nodes(tokens, tq)

## Get selected nodes from tokenindex
selected_nodes(tokens2)
```

`select_nodes` *Apply tquery to initiate reshape operations*

Description

Apply tquery to initiate reshape operations

Usage

```
select_nodes(
  tokens,
  tquery,
  fill = TRUE,
  fill_only_first = TRUE,
  .one_per_sentence = FALSE,
  .order = 1
)
```

Arguments

`tokens`	A tokenIndex data.table, or any data.frame coercible with as_tokenindex.
`tquery`	A tquery that selects and labels the nodes that are used in the reshape operations
`fill`	Logical, should fill be used?
`fill_only_first`	Logical, should a node only be filled once, with the nearest (first) labeled node?
`.one_per_sentence`	If true, only one match per sentence is used, giving priority to paterns closest to the root (or fartest from the root if .order = -1). This is sometimes necessary to deal with recursion.
`.order`	If .one_per_sentence is used, .order determines whether the paterns closest to (1) or farthest away (-1) are used.

Value

A tokenIndex with a .nodes attribute, that enables the use of reshape operations on the selected nodes

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text4',]

## use a tquery to label the nodes that you want to manipulate
tq = tquery(relation = "relcl", label = "relative_clause")

## apply query to select nodes
tokens2 = select_nodes(tokens, tq)

## as an example, we make the parent of the relative_clause
## nodes NA, effectively cutting of the relcl from the tree
tokens2 = mutate_nodes(tokens2, "relative_clause", parent=NA)

tokens2


if (interactive()) plot_tree(tokens2)

## this is designed to work nicely with magrittr piping
if (interactive()) {
tokens %>%
  select_nodes(tq) %>%
  mutate_nodes("relative_clause", parent=NA) %>%
  plot_tree()
}
```

---

set_rsyntax_threads *Set number of threads to be used by rsyntax functions*

Description

rsyntax relies heavily on the data.table package, which supports multithreading. By default, the number of threads set by data.table are used, as you can see with `getDTthreads`. Here you can set the number of threads for rsyntax functions, without affecting the data.table settings.

Usage

```
set_rsyntax_threads(threads = NULL)
```

Arguments

threads	The number of threads to use. Cannot be higher than number of threads used by data.table, which you can change with `setDTthreads`. If left empty (NULL), all data.table threads are used

Value

Does not return a value. Sets the global 'rsyntax_threads' option.

Examples

```
current_threads = rsyntax_threads()

set_rsyntax_threads(2)

## undo change (necessary for CRAN checks)
set_rsyntax_threads(current_threads)
```

`split_UD_conj` *Split conjunctions for dependency trees in Universal Dependencies*

Description

Split conjunctions for dependency trees in Universal Dependencies

Usage

```
split_UD_conj(
  tokens,
  conj_rel = "conj",
  cc_rel = c("cc", "cc:preconj"),
  unpack = T,
  no_fill = NULL,
  min_dist = 0,
  max_dist = Inf,
  right_fill_dist = T,
  compound_rel = c("compound*", "flat"),
  ...
)
```

Arguments

`tokens`	a tokenIndex based on texts parsed with `spacy_parse` (with dependency=TRUE)
`conj_rel`	The dependency relation for conjunctions. By default conj
`cc_rel`	The dependency relation for the coordinating conjunction. By default cc. This will be removed.
`unpack`	If TRUE (default), create separate branches for the parent and the node that inherits the parent position
`no_fill`	Optionally, a character vector with relation types that will be excluded from fill
`min_dist`	Optionally, a minimal distance between the conj node and its parent
`max_dist`	Optionally, a maximum distance between the conj node and its parent
`right_fill_dist`	Should fill to the right of the conjunction be used?
`compound_rel`	The relation types indicating compounds
`...`	specify conditions for the conjunction token. For instance, using 'pos = "VERB"' to only split VERB conjunctions. This is especially usefull to use different no_fill conditions.

Value

A tokenindex

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text5',]

if (interactive()) {
tokens %>%
   split_UD_conj() %>%
   plot_tree()
}
```

---

subset_nodes *Subset a select_nodes selection*

Description

Enables more control in reshape operations

Usage

```
subset_nodes(.tokens, subset, copy = TRUE)
```

Arguments

.tokens	A tokenIndex in which nodes are selected with select_nodes.
subset	A subset expression (that evaluates to a logical vector). The token column for each labeled node in the tquery can be referred to as label$column.
copy	If TRUE, make a deep copy of .tokens. Use if output does not overwrite .tokens

Value

A tokenIndex with a .nodes attribute

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text4',]

## use a tquery to label the nodes that you want to manipulate
tq = tquery(label='verb', children(relation='nsubj'))

## apply query to select nodes
tokens2 = select_nodes(tokens, tq)

selected_nodes(tokens2)$nodes
tokens2 = subset_nodes(tokens2, verb$relation == 'ROOT')
selected_nodes(tokens2)$nodes
```

syntax_reader *Create a full text browser with highlighted rsyntax annotations*

Description

Create a full text browser with highlighted rsyntax annotations

Usage

```
syntax_reader(
  tokens,
  annotation,
  value = NULL,
  value2 = NULL,
  meta = NULL,
  token_col = "token",
  filename = NULL,
  view = TRUE,
  random_seed = NA,
  ...
)
```

Arguments

tokens	A tokenIndex
annotation	The name of the column that contains the rsyntax annotation
value	Optionally, a character vector with values in annotation. If used, only these values are fully colored, and the other (non NA) values only have border colors.
value2	Optionally, a character vector with values in annotation other than those specified in 'value'. If used, only these values have border colors.
meta	Optionally, a data.frame with document meta data. Has to have a column named doc_id of which the values match with the doc_id column in tokens
token_col	The name of the column in tokens with the token text
filename	Optionally, a filename to directly save the file. If not specified, a temporary file is created
view	If TRUE, the browser will immediatly be viewed in the viewer panel
random_seed	If a number is given, it is used as a seed to randomize the order of documents. This is usefull for validations purposes, because the doc_id in the tokenindex is sorted.
...	Arguments passed to create_browser

Value

The url for the file

Examples

```
tokens = tokens_spacy

## two simple example tqueries
passive = tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("agent"), label = "subject"))
active =  tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("nsubj", "nsubjpass"), label = "subject"))



tokens = annotate_tqueries(tokens, 'clause', pas=passive, act=active)
syntax_reader(tokens, annotation = 'clause', value = 'subject')
```

---

tokens_corenlp *Example tokens for coreNLP English*

Description

Example tokens for coreNLP English

Usage

```
data(tokens_corenlp)
```

Format

data.frame

---

tokens_dutchclauses *Example tokens for Dutch clauses*

Description

Example tokens for Dutch clauses

Usage

```
data(tokens_dutchclauses)
```

Format

data.frame

tokens_dutchquotes *Example tokens for Dutch quotes*

Description

Example tokens for Dutch quotes

Usage

```
data(tokens_dutchquotes)
```

Format

data.frame

tokens_spacy *Example tokens for spacy English*

Description

Example tokens for spacy English

Usage

```
data(tokens_spacy)
```

Format

data.frame

tquery *Create a query for dependency based parse trees in a data.table (CoNLL-U or similar format).*

Description

To find nodes you can use named arguments, where the names are column names (in the data.table on which the queries will be used) and the values are vectors with look-up values.

Children or parents of nodes can be queried by passing the children or parents function as (named or unnamed) arguments. These functions use the same query format as the tquery function, and children and parents can be nested recursively to find children of children etc.

The custom_fill() function (also see fill argument) can be nested to customize which children of a 'labeled' node need to be matched. It can only be nested in a query if the label argument is not NULL, and by default will include all children of the node that have not been assigned to another node. If two nodes have a shared child, the child will be assigned to the closest node.

Please look at the examples below for a recommended syntactic style for using the find_nodes function and these nested functions.

Usage

```
tquery(..., g_id = NULL, label = NA, fill = TRUE, block = FALSE)
```

Arguments

`...`	Accepts two types of arguments: name-value pairs for finding nodes (i.e. rows), and functions to look for parents/children of these nodes. The name in the name-value pairs need to match a column in the data.table, and the value needs to be a vector of the same data type as the column. By default, search uses case sensitive matching, with the option of using common wildcards (* for any number of characters, and ? for a single character). Alternatively, flags can be used to to change this behavior to 'fixed' (__F), 'igoring case' (__I) or 'regex' (__R). See details for more information. If multiple name-value pairs are given, they are considered as AND statements, but see details for syntax on using OR statements, and combinations. To look for parents and children of the nodes that are found, you can use the parents and children functions as (named or unnamed) arguments. These functions have the same query arguments as tquery, but with some additional arguments.
`g_id`	Find nodes by global id, which is the combination of the doc_id, sentence and token_id. Passed as a data.frame or data.table with 3 columns: (1) doc_id, (2) sentence and (3) token_id.
`label`	A character vector, specifying the column name under which the selected tokens are returned. If NA, the column is not returned.
`fill`	Logical. If TRUE (default), the default custom_fill() will be used. To more specifically control fill, you can nest the custom_fill function (a special version of the children function).
`block`	Logical. If TRUE, the node will be blocked from being assigned (labeled). This is mainly useful if you have a node that you do not want to be assigned by fill, but also don't want to 'label' it. Essentially, block is shorthand for using label and then removing the node afterwards. If block is TRUE, label has to be NA.

Details

Multiple values in a name-value pair operate as OR conditions. For example, tquery(relation = c('nsubj','dobj')) means that the relation column should have the value 'nsubj' OR 'dobj'.

If multiple named arguments are given they operate as AND conditions. For example, tquery(relation = 'nsubj', pos = 'PROPN') means that the relation should be 'nsubj' AND the pos should be 'PROPN'.

This easily combines for the most common use case, which is to select on multiple conditions (relation AND pos), but allowing different (similar) values ('PROPN' OR 'NOUN'). For example: tquery(relation = 'nsubj', pos = c('PROPN','NOUN')) means that the node should have the 'nsubj' relation, but pos can be either 'PROPN' or 'NOUN'.

For more specific behavior, the AND(), OR() and NOT() functions can be used for boolean style conditions.

There are several flags that can be used to change search condition. To specify flags, add a double underscore and the flag character to the name in the name value pairs (...). By adding the suffix __R, query terms are considered to be regular expressions, and the suffix __I uses case insensitive search (for normal or regex search). If the suffix __F is used, only exact matches are valid (case sensitive, and no wildcards). Multiple flags can be combined, such as lemma__RI, or lemma_IR (order of flags is irrelevant)

Value

A tQuery object, that can be used with the apply_queries function.

Examples

```
## it is convenient to first prepare vectors with relevant words/pos-tags/relations
.SAY_VERBS = c("tell", "show","say", "speak") ## etc.
.QUOTE_RELS=  c("ccomp", "dep", "parataxis", "dobj", "nsubjpass", "advcl")
.SUBJECT_RELS = c('su', 'nsubj', 'agent', 'nmod:agent')

quotes_direct = tquery(lemma = .SAY_VERBS,
                       children(label = 'source', p_rel = .SUBJECT_RELS),
                       children(label = 'quote', p_rel = .QUOTE_RELS))
quotes_direct
```

---

unselect_nodes *Undo select_nodes*

Description

Not strictly required. Only available for elegance and minor memory efficiency

Usage

```
unselect_nodes(.tokens)
```

Arguments

.tokens	A tokenIndex in which nodes are selected with select_nodes.

Value

A tokenIndex (without a .nodes attribute)

Examples

```
tokens = tokens_spacy[tokens_spacy$doc_id == 'text4',]

tq = tquery(relation = "relcl", label = "relative_clause")
tokens = select_nodes(tokens, tq)
selected_nodes(tokens)

tokens = unselect_nodes(tokens)

is.null(attr(tokens, '.nodes'))
```

Index