Package 'NLP'

August 23, 2014

Version 0.1-5

Title Natural Language Processing Infrastructure

Description Basic classes and methods for Natural Language Processing.

License GPL-2

Imports utils

Author Kurt Hornik [aut, cre]

Maintainer Kurt Hornik <Kurt.Hornik@R-project.org>

Depends R (>= 2.10)

NeedsCompilation no

Repository CRAN

Date/Publication 2014-08-23 10:35:56

R topics documented:

annotate 2
AnnotatedPlainTextDocument 3
Annotation 5
annotations_in_spans 7
Annotator 8
annotators 10
CoNLLTextDocument 13
datetime 14
generics 15
language 16
Span 17
String 18
TaggedTextDocument 20
tagsets 21
TextDocument 21

tokenizers 22
Tree 23
utils 25
viewers 26
WordListDocument 27

Index **28**

`annotate` *Annotate text strings*

Description

Compute annotations by iteratively calling the given annotators with the given text and current annotations, and merging the newly computed annotations with the current ones.

Usage

```
annotate(s, f, a = Annotation())
```

Arguments

`s`	a `String` object, or something coercible to this using `as.String` (e.g., a character string with appropriate encoding information)
`f`	an `Annotator` object, or a list of such objects representing an annotator pipeline.
`a`	an `Annotation` object giving the annotations to start with.

Value

An annotation object containing the iteratively computed and merged annotations.

Examples

```
## A simple text.
s <- String("  First sentence.  Second sentence.  ")
##           ****5****0****5****0****5****0****5**

## A very trivial sentence tokenizer.
sent_tokenizer <-
function(s) {
    s <- as.String(s)
    m <- gregexpr("[^[:space:]][^.]*\\.", s)[[1L]]
    Span(m, m + attr(m, "match.length") - 1L)
}
## (Could also use Regexp_Tokenizer() with the above regexp pattern.)
## A simple sentence token annotator based on the sentence tokenizer.
sent_token_annotator <- Simple_Sent_Token_Annotator(sent_tokenizer)

## Annotate sentence tokens.
```

```
a1 <- annotate(s, sent_token_annotator)
a1

## A very trivial word tokenizer.
word_tokenizer <-
function(s) {
    s <- as.String(s)
    ## Remove the last character (should be a period when using
    ## sentences determined with the trivial sentence tokenizer).
    s <- substring(s, 1L, nchar(s) - 1L)
    ## Split on whitespace separators.
    m <- gregexpr("[^[:space:]]+", s)[[1L]]
    Span(m, m + attr(m, "match.length") - 1L)
}
## A simple word token annotator based on the word tokenizer.
word_token_annotator <- Simple_Word_Token_Annotator(word_tokenizer)

## Annotate word tokens using the already available sentence token
## annotations.
a2 <- annotate(s, word_token_annotator, a1)
a2

## Can also perform sentence and word token annotations in a pipeline:
annotate(s, list(sent_token_annotator, word_token_annotator))
```

`AnnotatedPlainTextDocument`

Annotated Plain Text Documents

Description

Create annotated plain text documents from plain text and collections of annotations for this text.

Usage

```
AnnotatedPlainTextDocument(x, annotations, meta = list())
annotations(x)
```

Arguments

`x`	For `AnnotatedPlainTextDocument()`, a `String` object, or something coercible to this using `as.String` (e.g., a character string with appropriate encoding information). For `annotations()`, an object inheriting from class `"AnnotatedPlainTextDocument"`.
`annotations`	an `Annotation` object with annotations for `x`, or a list of such objects.
`meta`	a named or empty list of document metadata tag-value pairs.

Details

Annotated plain text documents combine plain text with collections ("sets", implemented as lists) of objects with annotations for the text.

A typical workflow is to use `annotate` with suitable annotator pipelines to obtain the annotations, and then combine these with the text being annotated using `AnnotatedPlainTextDocument()`, which returns an object inheriting from `"AnnotatedPlainTextDocument"` and `"TextDocument"`, from which the text and collection of annotations can be obtained using, respectively, `as.character` and `annotations`.

There are methods for generics `words`, `sents`, `paras`, `tagged_words`, `tagged_sents`, `tagged_paras`, `chunked_sents`, `parsed_sents` and `parsed_paras` and class `"AnnotatedPlainTextDocument"` providing structured views of the text in such documents. These all have an additional argument `which` for specifying the annotation object to use (by default, the first one is taken), and of course require the necessary annotations to be available in the annotation object used.

Value

For `AnnotatedPlainTextDocument()`, an object inheriting from `"AnnotatedPlainTextTextDocument"` and `"TextDocument"`.

For `annotations()`, a list of `Annotation` objects.

See Also

`TextDocument` for basic information on the text document infrastructure employed by package **NLP**.

Examples

```
## Use a pre-built annotated plain text document obtained by employing an
## annotator pipeline from package 'StanfordCoreNLP', available from the
## repository at <http://datacube.wu.ac.at>, using the following code:
##   require("StanfordCoreNLP")
##   x <- paste("Stanford University is located in California.",
##              "It is a great university.")
##   p <- StanfordCoreNLP_Pipeline(c("pos", "lemma", "parse"))
##   doc <- AnnotatedPlainTextDocument(x, p(x))

doc <- readRDS(system.file("texts", "stanford.rds", package = "NLP"))

doc

## Extract available annotation:
a <- annotations(doc)[[1L]]
a

## Structured views:
sents(doc)
tagged_sents(doc)
parsed_sents(doc)

## Add (trivial) paragraph annotation:
```

```
s <- as.character(doc)
a <- annotate(s, Simple_Para_Token_Annotator(blankline_tokenizer), a)
doc <- AnnotatedPlainTextDocument(s, a)
## Structured view:
paras(doc)
```

Annotation *Annotation objects*

Description

Creation and manipulation of annotation objects.

Usage

```
Annotation(id = NULL, type = NULL, start, end, features = NULL)
as.Annotation(x, ...)
## S3 method for class 'Span'
as.Annotation(x, id = NULL, type = NULL, ...)
is.Annotation(x)
```

Arguments

id	an integer vector giving the annotation ids, or NULL (default) resulting in missing ids.
type	a character vector giving the annotation types, or NULL (default) resulting in missing types.
start, end	integer vectors giving the start and end positions of the character spans the annotations refer to.
features	a list of (named or empty) features lists, or NULL (default), resulting in empty feature lists.
x	an R object (an object of class "Span" for the coercion methods for such objects).
...	further arguments passed to or from other methods.

Details

A single annotation (of natural language text) is a quintuple with "slots" 'id', 'type', 'start', 'end', and 'features'. These give, respectively, id and type, the character span the annotation refers to, and a collection of annotation features (tag/value pairs).

Annotation objects provide sequences (allowing positional access) of single annotations. They have class "Annotation" and, as they contain character spans, also inherit from class "Span". Span objects can be coerced to annotation objects via as.Annotation which allows to specify ids and types (using the default values sets these to missing), and annotation objects can be coerced to span objects using as.Span.

The features of a single annotation are represented as named or empty lists.

Subscripting annotation objects via `[` extracts subsets of annotations; subscripting via `$` extracts the sequence of values of the named slot, i.e., an integer vector for 'id', 'start', and 'end', a character vector for 'type', and a list of named or empty lists for 'features'.

There are several additional methods for class `"Annotation"`: `print` and `format` (which both have a `values` argument which if `FALSE` suppresses indicating the feature map values); `c` combines annotations; `merge` merges annotations by combining the feature lists of annotations with otherwise identical slots; `subset` allows subsetting by expressions involving the slot names; and `as.list` and `as.data.frame` coerce, respectively, to lists (of single annotation objects) and data frames (with annotations and slots corresponding to rows and columns).

`Annotation()` creates annotation objects from the given sequences of slot values: those not `NULL` must all have the same length (the number of annotations in the object).

`as.Annotation()` coerces to annotation objects, with a method for span objects.

`is.Annotation()` tests whether an object inherits from class `"Annotation"`.

Value

For `Annotation()` and `as.Annotation()`, an annotation object (of class `"Annotation"` also inheriting from class `"Span"`).

For `is.Annotation`, a logical.

Examples

```
## A simple text.
s <- String("  First sentence.  Second sentence.  ")
##           ****5****0****5****0****5****0****5**

## Basic sentence and word token annotations for the text.
a1s <- Annotation(1 : 2,
                  rep.int("sentence", 2L),
                  c( 3L, 20L),
                  c(17L, 35L))
a1w <- Annotation(3 : 6,
                  rep.int("word", 4L),
                  c( 3L,  9L, 20L, 27L),
                  c( 7L, 16L, 25L, 34L))

## Use c() to combine these annotations:
a1 <- c(a1s, a1w)
a1
## Subscripting via '[':
a1[3 : 4]
## Subscripting via '$':
a1$type
## Subsetting according to slot values, directly:
a1[a1$type == "word"]
## or using subset():
subset(a1, type == "word")
```

```
## We can subscript string objects by annotation objects to extract the
## annotated substrings:
s[subset(a1, type == "word")]
## We can also subscript by lists of annotation objects:
s[annotations_in_spans(subset(a1, type == "word"),
                       subset(a1, type == "sentence"))]

## Suppose we want to add the sentence constituents (the ids of the
## words in the respective sentences) to the features of the sentence
## annotations.  The basic computation is
lapply(annotations_in_spans(a1[a1$type == "word"],
                            a1[a1$type == "sentence"]),
       function(a) a$id)
## For annotations, we need lists of feature lists:
features <-
    lapply(annotations_in_spans(a1[a1$type == "word"],
                                a1[a1$type == "sentence"]),
           function(e) list(constituents = e$id))
## Could add these directly:
a2 <- a1
a2$features[a2$type == "sentence"] <- features
a2
## Note how the print() method summarizes the features.
## We could also write a sentence constituent annotator
## (note that annotators should always have formals 's' and 'a', even
## though for computing the sentence constituents s is not needed):
sent_constituent_annotator <-
Annotator(function(s, a) {
              i <- which(a$type == "sentence")
              features <-
                  lapply(annotations_in_spans(a[a$type == "word"],
                                              a[i]),
                         function(e) list(constituents = e$id))
              Annotation(a$id[i], a$type[i], a$start[i], a$end[i],
                         features)
          })
sent_constituent_annotator(s, a1)
## Can use merge() to merge the annotations:
a2 <- merge(a1, sent_constituent_annotator(s, a1))
a2
## Equivalently, could have used
a2 <- annotate(s, sent_constituent_annotator, a1)
a2
## which merges automatically.
```

`annotations_in_spans` *Annotations contained in character spans*

Description

Extract annotations contained in character spans.

Usage

```
annotations_in_spans(x, y)
```

Arguments

x	an Annotation object.
y	a Span object, or something coercible to this (such as an Annotation object)

Value

A list with elements the annotations in x with character spans contained in the respective elements of y.

Examples

```
## A simple text.
s <- String("  First sentence.  Second sentence.  ")
##           ****5****0****5****0****5****0****5**

## Basic sentence and word token annotation for the text.
a <- c(Annotation(1 : 2,
                  rep.int("sentence", 2L),
                  c( 3L, 20L),
                  c(17L, 35L)),
       Annotation(3 : 6,
                  rep.int("word", 4L),
                  c( 3L,  9L, 20L, 27L),
                  c( 7L, 16L, 25L, 34L)))

## Annotation for word tokens according to sentence:
annotations_in_spans(a[a$type == "word"], a[a$type == "sentence"])
```

Annotator *Annotator objects*

Description

Create annotator objects.

Usage

```
Annotator(f, description = NULL, classes = NULL)
```

Arguments

f	an annotator function, which must have formals s and a giving, respectively, the string with the natural language text to annotate and an annotation object to start from, and return an annotation object with the computed annotations.
description	a character string describing the annotator, or NULL (default).
classes	a character vector or NULL (default) giving additional classes to be used for the created annotator object in addition to "Annotator".

Details

Annotator() checks that the given annotator function has the appropriate formals, and returns an annotator object which inherits from the given classes and "Annotator", and contains the given description (currently, as an attribute) to be used in the print method for such objects.

Value

An annotator object inheriting from the given classes and class "Annotator".

See Also

Simple annotator generators for creating "simple" annotator objects based on function performing simple basic NLP tasks.

Package **StanfordCoreNLP** available from the repository at http://datacube.wu.ac.at which provides annotator generators for annotator pipelines based on the Stanford CoreNLP tools.

Examples

```
## Use blankline_tokenizer() for a simple paragraph token annotator:
para_token_annotator <-
Annotator(function(s, a = Annotation()) {
              spans <- blankline_tokenizer(s)
              n <- length(spans)
              ## Need n consecutive ids, starting with the next "free"
              ## one:
              from <- next_id(a$id)
              Annotation(seq(from = from, length.out = n),
                         rep.int("paragraph", n),
                         spans$start,
                         spans$end)
          },
          "A paragraph token annotator based on blankline_tokenizer().")
para_token_annotator
## Alternatively, use Simple_Para_Token_Annotator().

## A simple text with two paragraphs:
s <- String(paste("  First sentence.  Second sentence.  ",
                  "  Second paragraph.  ",
                  sep = "\n\n"))
a <- annotate(s, para_token_annotator)
## Annotations for paragraph tokens.
```

```
a
## Extract paragraph tokens.
s[a]
```

---

annotators *Simple annotator generators*

Description

Create annotator objects for composite basic NLP tasks based on functions performing simple basic tasks.

Usage

```
Simple_Para_Token_Annotator(f, description = NULL, classes = NULL)
Simple_Sent_Token_Annotator(f, description = NULL, classes = NULL)
Simple_Word_Token_Annotator(f, description = NULL, classes = NULL)
Simple_POS_Tag_Annotator(f, description = NULL, classes = NULL)
Simple_Entity_Annotator(f, description = NULL, classes = NULL)
Simple_Chunk_Annotator(f, description = NULL, classes = NULL)
Simple_Stem_Annotator(f, description = NULL, classes = NULL)
```

Arguments

`f`	a function performing a "simple" basic NLP task (see **Details**).
`description`	a character string describing the annotator, or `NULL` (default).
`classes`	a character vector or `NULL` (default) giving additional classes to be used for the created annotator object in addition to the default ones (see **Details**).

Details

The purpose of these functions is to facilitate the creation of annotators for basic NLP tasks as described below.

`Simple_Para_Token_Annotator()` creates "simple" paragraph token annotators. Argument `f` should be a paragraph tokenizer, which takes a string `s` with the whole text to be processed, and returns the spans of the paragraphs in `s`, or an annotation object with these spans and (possibly) additional features. The generated annotator inherits from the default classes `"Simple_Para_Token_Annotator"` and `"Annotator"`. It uses the results of the simple paragraph tokenizer to create and return annotations with unique ids and type 'paragraph'.

`Simple_Sent_Token_Annotator()` creates "simple" sentence token annotators. Argument `f` should be a sentence tokenizer, which takes a string `s` with the whole text to be processed, and returns the spans of the sentences in `s`, or an annotation object with these spans and (possibly) additional features. The generated annotator inherits from the default classes `"Simple_Sent_Token_Annotator"` and `"Annotator"`. It uses the results of the simple sentence tokenizer to create and return annotations with unique ids and type 'sentence', possibly combined with sentence constituent features for already available paragraph annotations.

`Simple_Word_Token_Annotator()` creates "simple" word token annotators. Argument `f` should be a simple word tokenizer, which takes a string `s` giving a sentence to be processed, and returns the spans of the word tokens in `s`, or an annotation object with these spans and (possibly) additional features. The generated annotator inherits from the default classes `"Simple_Word_Token_Annotator"` and `"Annotator"`. It uses already available sentence token annotations to extract the sentences and obtains the results of the word tokenizer for these. It then adds the sentence character offsets and unique word token ids, and word token constituents features for the sentences, and returns the word token annotations combined with the augmented sentence token annotations.

`Simple_POS_Tag_Annotator()` creates "simple" POS tag annotators. Argument `f` should be a simple POS tagger, which takes a character vector giving the word tokens in a sentence, and returns either a character vector with the tags, or a list of feature maps with the tags as 'POS' feature and possibly other features. The generated annotator inherits from the default classes `"Simple_POS_Tag_Annotator"` and `"Annotator"`. It uses already available sentence and word token annotations to extract the word tokens for each sentence and obtains the results of the simple POS tagger for these, and returns annotations for the word tokens with the features obtained from the POS tagger.

`Simple_Entity_Annotator()` creates "simple" entity annotators. Argument `f` should be a simple entity detector ("named entity recognizer") which takes a character vector giving the word tokens in a sentence, and return an annotation object with the *word* token spans, a 'kind' feature giving the kind of the entity detected, and possibly other features. The generated annotator inherits from the default classes `"Simple_Entity_Annotator"` and `"Annotator"`. It uses already available sentence and word token annotations to extract the word tokens for each sentence and obtains the results of the simple entity detector for these, transforms word token spans to character spans and adds unique ids, and returns the combined entity annotations.

`Simple_Chunk_Annotator()` creates "simple" chunk annotators. Argument `f` should be a simple chunker, which takes as arguments character vectors giving the word tokens and the corresponding POS tags, and returns either a character vector with the chunk tags, or a list of feature lists with the tags as 'chunk_tag' feature and possibly other features. The generated annotator inherits from the default classes `"Simple_Chunk_Annotator"` and `"Annotator"`. It uses already available annotations to extract the word tokens and POS tags for each sentence and obtains the results of the simple chunker for these, and returns word token annotations with the chunk features (only).

`Simple_Stem_Annotator()` creates "simple" stem annotators. Argument `f` should be a simple stemmer, which takes as arguments a character vector giving the word tokens, and returns a character vector with the corresponding word stems. The generated annotator inherits from the default classes `"Simple_Stem_Annotator"` and `"Annotator"`. It uses already available annotations to extract the word tokens, and returns word token annotations with the corresponding stem features (only).

In all cases, if the underlying simple processing function returns annotation objects these should not provide their own ids (or use such in the features), as the generated annotators will necessarily provide these (the already available annotations are only available at the annotator level, but not at the simple processing level).

Value

An annotator object inheriting from the given classes and the default ones.

See Also

Package **openNLP** which provides annotator generators for sentence and word tokens, POS tags, entities and chunks, using processing functions based on the respective Apache OpenNLP MaxEnt processing resources.

Examples

```
## A simple text.
s <- String("  First sentence.  Second sentence.  ")
##           ****5****0****5****0****5****0****5**

## A very trivial sentence tokenizer.
sent_tokenizer <-
function(s) {
    s <- as.String(s)
    m <- gregexpr("[^[:space:]][^.]*\\.", s)[[1L]]
    Span(m, m + attr(m, "match.length") - 1L)
}
## (Could also use Regexp_Tokenizer() with the above regexp pattern.)
sent_tokenizer(s)
## A simple sentence token annotator based on the sentence tokenizer.
sent_token_annotator <- Simple_Sent_Token_Annotator(sent_tokenizer)
sent_token_annotator
a1 <- annotate(s, sent_token_annotator)
a1
## Extract the sentence tokens.
s[a1]

## A very trivial word tokenizer.
word_tokenizer <-
function(s) {
    s <- as.String(s)
    ## Remove the last character (should be a period when using
    ## sentences determined with the trivial sentence tokenizer).
    s <- substring(s, 1L, nchar(s) - 1L)
    ## Split on whitespace separators.
    m <- gregexpr("[^[:space:]]+", s)[[1L]]
    Span(m, m + attr(m, "match.length") - 1L)
}
lapply(s[a1], word_tokenizer)
## A simple word token annotator based on the word tokenizer.
word_token_annotator <- Simple_Word_Token_Annotator(word_tokenizer)
word_token_annotator
a2 <- annotate(s, word_token_annotator, a1)
a2
## Extract the word tokens.
s[subset(a2, type == "word")]

## A simple word token annotator based on wordpunct_tokenizer():
word_token_annotator <-
    Simple_Word_Token_Annotator(wordpunct_tokenizer,
                                "Based on wordpunct_tokenizer().")
```

```
word_token_annotator
a2 <- annotate(s, word_token_annotator, a1)
a2
## Extract the word tokens.
s[subset(a2, type == "word")]
```

CoNLLTextDocument *CoNLL-Style Text Documents*

Description

Create text documents from CoNLL-style files.

Usage

```
CoNLLTextDocument(con, encoding = "unknown", meta = list())
```

Arguments

con	a connection object or a character string. See scan for details.
encoding	encoding to be assumed for input strings. See scan for details.
meta	a named or empty list of document metadata tag-value pairs.

Details

CoNLL-style files use an extended tabular format where empty lines separate sentences, and non-empty lines consist of whitespace separated columns giving the word tokens and annotations for these. In principle, these annotations can vary from corpus to corpus: the current version of CoNLLTextDocument() assumes a fixed set of 3 columns giving, respectively, the word token and its POS and chunk tags.

The lines are read from the given connection and split into fields using scan(). From this, a suitable representation of the provided information is obtained, and returned as a CoNLL text document object inheriting from classes "CoNLLTextDocument" and "TextDocument".

There are methods for generics words, sents, tagged_words, tagged_sents, and chunked_sents (as well as as.character) and class "CoNLLTextDocument", which should be used to access the text in such text document objects.

Value

An object inheriting from "CoNLLTextDocument" and "TextDocument".

See Also

TextDocument for basic information on the text document infrastructure employed by package **NLP**.

http://ifarm.nl/signll/conll/ for general information about CoNLL (Conference on Natural Language Learning), the yearly meeting of the Special Interest Group on Natural Language Learning of the Association for Computational Linguistics.

http://www.cnts.ua.ac.be/conll2000/chunking/ for the CoNLL 2000 chunking task, and training and test data sets which can be read in using `CoNLLTextDocument()`.

---

datetime *Parse ISO 8601 Date/Time Strings*

Description

Extract date/time components from strings following one of the six formats specified in the NOTE-datetime ISO 8601 profile (http://www.w3.org/TR/NOTE-datetime).

Arguments

x	a character vector.

Details

For character strings in one of the formats in the profile, the corresponding date/time components are extracted, with seconds and decimal fractions of seconds combined. Other (malformed) strings are warned about.

The extracted components for each string are gathered into a named list with elements of the appropriate type (integer for year to min; double for sec; character for the time zone designator). The object returned is a (suitably classed) list of such named lists. This internal representation may change in future versions.

One can subscript such ISO 8601 date/time objects using `[` and extract components using `$` (where missing components will result in `NA`s), and convert them to the standard R date/time classes using `as.Date`, `as.POSIXct` and `as.POSIXlt` (incomplete elements will convert to suitably missing elements). In addition, there are `print` and `as.data.frame` methods for such objects.

Value

An object inheriting from class `"ISO_8601_datetime"` with the extracted date/time components.

Examples

```
## Use the examples from <http://www.w3.org/TR/NOTE-datetime>, plus one
## in UTC.
x <- c("1997",
       "1997-07",
       "1997-07-16",
```

```
       "1997-07-16T19:20+01:00",
       "1997-07-16T19:20:30+01:00",
       "1997-07-16T19:20:30.45+01:00",
       "1997-07-16T19:20:30.45Z")
y <- parse_ISO_8601_datetime(x)
y
## Conversions: note that "incomplete" elements are converted to
## "missing".
as.Date(y)
as.POSIXlt(y)
## Subscripting and extracting components:
head(y, 3)
y$mon
```

`generics` *Access or Modify Content or Metadata*

Description

Access or modify the content or metadata of R objects.

Usage

```
content(x)
content(x) <- value
meta(x, tag = NULL, ...)
meta(x, tag = NULL, ...) <- value
```

Arguments

`x`	an R object.
`value`	a suitable R object.
`tag`	a character string or `NULL` (default), indicating to return the single metadata value for the given tag, or all metadata tag/value pairs.
`...`	arguments to be passed to or from methods.

Details

These are generic functions, with no default methods.

Often, classed R objects (e.g., those representing text documents in packages **NLP** and **tm**) contain information that can be grouped into "content", metadata and other components, where content can be arbitrary, and metadata are collections of tag/value pairs represented as named or empty lists. The `content()` and `meta()` getters and setters aim at providing a consistent high-level interface to the respective information (abstracting from how classes internally represent the information).

Value

Methods for `meta()` should return a named or empty list of tag/value pairs if no tag is given (default), or the value for the given tag.

See Also

`TextDocument` for basic information on the text document infrastructure employed by package **NLP**.

`language` *Parse IETF Language Tag*

Description

Extract language, script, region and variant subtags from IETF language tags.

Usage

```
parse_IETF_language_tag(x, expand = FALSE)
```

Arguments

`x`	a character vector with IETF language tags.
`expand`	a logical indicating whether to expand subtags into their description(s).

Details

Internet Engineering Task Force (IETF) language tags are defined by IETF BCP 47, which is currently composed by the normative RFC 5646 (`http://tools.ietf.org/html/rfc5646`) and RFC 4647 (`http://tools.ietf.org/html/rfc4646`), along with the normative content of the IANA Language Subtag Registry regulated by these RFCs. These tags are used in a number of modern computing standards.

Each language tag is composed of one or more "subtags" separated by hyphens. Normal language tags have the following subtags:

- a language subtag (optionally, with language extension subtags),
- an optional script subtag,
- an optional region subtag,
- optional variant subtags,
- optional extension subtags,
- an optional private use subtag.

Language subtags are mainly derived from ISO 639-1 and ISO 639-2, script subtags from ISO 15924, and region subtags from ISO 3166-1 alpha-2 and UN M.49. (See package **ISOcodes** for more information about these standards.) Variant subtags are not derived from any standard. The Language Subtag Registry (http://www.iana.org/assignments/language-subtag-registry), maintained by the Internet Assigned Numbers Authority (IANA), lists the current valid public subtags, as well as the so-called "grandfathered" language tags.

See http://en.wikipedia.org/wiki/IETF_language_tag for more information.

Value

If expand is false, a list of character vectors of the form *"type=subtag"*, where *type* gives the type of the corresponding subtag (one of 'Language', 'Extlang', 'Script', 'Region', 'Variant', or 'Extension'), or *"type=tag"* with *type* either 'Privateuse' or 'Grandfathered'.

Otherwise, a list of lists of character vectors obtained by replacing the subtags by their corresponding descriptions (which may be multiple) from the IANA registry. Note that no such descriptions for Extension and Privateuse subtags are available in the registry; on the other hand, empty expansions of the other subtags indicate malformed tags (as these subtags must be available in the registry).

Examples

```
## German as used in Switzerland:
parse_IETF_language_tag("de-CH")
## Serbian written using Latin script as used in Serbia and Montenegro:
parse_IETF_language_tag("sr-Latn-CS")
## Spanish appropriate to the UN Latin American and Caribbean region:
parse_IETF_language_tag("es-419")
## All in one:
parse_IETF_language_tag(c("de-CH", "sr-Latn-CS", "es-419"))
parse_IETF_language_tag(c("de-CH", "sr-Latn-CS", "es-419"),
                        expand = TRUE)
## Two grandfathered tags:
parse_IETF_language_tag(c("i-klingon", "zh-min-nan"),
                        expand = TRUE)
```

Span *Span objects*

Description

Creation and manipulation of span objects.

Usage

```
Span(start, end)
as.Span(x)
is.Span(x)
```

Arguments

`start, end`	integer vectors giving the start and end positions of the spans.
`x`	an R object.

Details

A single span is a pair with "slots" 'start' and 'end', giving the start and end positions of the span.

Span objects provide sequences (allowing positional access) of single spans. They have class `"Span"`. Span objects can be coerced to annotation objects via `as.Annotation` (which of course is only appropriate provided that the spans are character spans of the natural language text being annotated), and annotation objects can be coerced to span objects via `as.Span` (giving the character spans of the annotations).

Subscripting span objects via `[` extracts subsets of spans; subscripting via `$` extracts integer vectors with the sequence of values of the named slot.

There are several additional methods for class `"Span"`: `print` and `format`; `c` combines spans, and `as.list` and `as.data.frame` coerce, respectively, to lists (of single span objects) and data frames (with spans and slots corresponding to rows and columns). Finally, one can add a scalar and a span object (resulting in shifting the start and end positions by the scalar).

`Span()` creates span objects from the given sequence of start and end positions, which must have the same length.

`as.Span()` coerces to span objects, with a method for annotation objects.

`is.Span()` tests whether an object inherits from class `"Span"` (and hence returns `TRUE` for both span and annotation objects).

Value

For `Span()` and `as.Span()`, a span object (of class `"Span"`).

For `is.Span()`, a logical.

`String` *String objects*

Description

Creation and manipulation of string objects.

Usage

```
String(x)
as.String(x)
is.String(x)
```

Arguments

x	a character vector with the appropriate encoding information for String; an arbitrary R object otherwise.

Details

String objects provide character strings encoded in UTF-8 with class "String", which currently has a useful [subscript method: with indices i and j of length one, this gives a string object with the substring starting at the position given by i and ending at the position given by j; subscripting with a single index which is an object inheriting from class "Span" or a list of such objects returns a character vector of substrings with the respective spans, or a list thereof.

Further methods may be added in the future.

String() creates a string object from a given character vector, taking the first element of the vector and converting it to UTF-8 encoding.

as.String() coerces to a string object by calling String on the result of the toString generic.

is.String() tests whether an object inherits from class "String".

Value

For String() and as.String(), a string object (of class "String").

For is.String(), a logical.

Examples

```
## A simple text.
s <- String("  First sentence.  Second sentence.  ")
##           ****5****0****5****0****5****0****5**

## Basic sentence and word token annotation for the text.
a <- c(Annotation(1 : 2,
                  rep.int("sentence", 2L),
                  c( 3L, 20L),
                  c(17L, 35L)),
       Annotation(3 : 6,
                  rep.int("word", 4L),
                  c( 3L,  9L, 20L, 27L),
                  c( 7L, 16L, 25L, 34L)))

## All word tokens (by subscripting with an annotation object):
s[a[a$type == "word"]]
## Word tokens according to sentence (by subscripting with a list of
## annotation objects):
s[annotations_in_spans(a[a$type == "word"], a[a$type == "sentence"])]
```

`TaggedTextDocument` *POS-Tagged Word Text Documents*

Description

Create text documents from files containing POS-tagged words.

Usage

```
TaggedTextDocument(con, encoding = "unknown",
                   word_tokenizer = whitespace_tokenizer,
                   sent_tokenizer = Regexp_Tokenizer("\n", invert = TRUE),
                   para_tokenizer = blankline_tokenizer,
                   sep = "/",
                   meta = list())
```

Arguments

`con`	a connection object or a character string. See `readLines` for details.
`encoding`	encoding to be assumed for input strings. See `readLines` for details.
`word_tokenizer`	a function for obtaining the word token spans.
`sent_tokenizer`	a function for obtaining the sentence token spans.
`para_tokenizer`	a function for obtaining the paragraph token spans, or `NULL` in which case no paragraph tokenization is performed.
`sep`	the character string separating the word tokens and their POS tags.
`meta`	a named or empty list of document metadata tag-value pairs.

Details

`TaggedTextDocument()` creates documents representing natural language text as suitable collections of POS-tagged words, based on using `readLines()` to read text lines from connections providing such collections.

The text read is split into paragraph, sentence and tagged word tokens using the span tokenizers specified by arguments `para_tokenizer`, `sent_tokenizer` and `word_tokenizer`. By default, paragraphs are assumed to be separated by blank lines, sentences by newlines and tagged word tokens by whitespace. Finally, word tokens and their POS tags are obtained by splitting the tagged word tokens according to `sep`. From this, a suitable representation of the provided collection of POS-tagged words is obtained, and returned as a tagged text document object inheriting from classes `"TaggedTextDocument"` and `"TextDocument"`.

There are methods for generics `words`, `sents`, `paras`, `tagged_words`, `tagged_sents`, and `tagged_paras` (as well as `as.character`) and class `"TaggedTextDocument"`, which should be used to access the text in such text document objects.

Value

An object inheriting from `"TaggedTextDocument"` and `"TextDocument"`.

See Also

http://nltk.github.com/nltk_data/packages/corpora/brown.zip which provides the W. N. Francis and H. Kucera Brown tagged word corpus as an archive of files which can be read in using `TaggedWordDocument()`.

---

tagsets *NLP Tag Sets*

Description

Tag sets frequently used in Natural Language Processing.

Usage

```
Penn_Treebank_POS_tags
```

Details

`Penn_Treebank_POS_tags` provides the Penn Treebank POS tags (http://www.cis.upenn.edu/~treebank) as a data frame with the following variables:

entry a character vector with the POS tags

description a character vector with short descriptions of the tags

examples a character vector with examples for the tags

Examples

```
Penn_Treebank_POS_tags
```

TextDocument *Text Documents*

Description

Representing and computing on text documents.

Details

Text documents are documents containing (natural language) text. In packages which employ the infrastructure provided by package **NLP**, such documents are represented via the virtual S3 class `"TextDocument"`: such packages then provide S3 text document classes extending the virtual base class (such as the `AnnotatedPlainTextDocument` objects provided by package **NLP** itself).

All extension classes must provide an `as.character` method which extracts the natural language text in documents of the respective classes in a "suitable" (not necessarily structured) form, as well as `content` and `meta` methods for accessing the (possibly raw) document content and metadata.

In addition, the infrastructure features the generic functions `words()`, `sents()`, etc., for which extension classes can provide methods giving a structured view of the text contained in documents of these classes (returning, e.g., a character vector with the word tokens in these documents, and a list of such character vectors).

See Also

`AnnotatedPlainTextDocument`, `CoNLLTextDocument`, `TaggedTextDocument`, and `WordListDocument` for the text document classes provided by package **NLP**.

`tokenizers` *Regexp tokenizers*

Description

Tokenizers using regular expressions to match either tokens or separators between tokens.

Usage

```
Regexp_Tokenizer(pattern, description = NULL, invert = FALSE, ...)
blankline_tokenizer(s)
whitespace_tokenizer(s)
wordpunct_tokenizer(s)
```

Arguments

`pattern`	a character string giving the regular expression to use for matching.
`description`	a character string describing the tokenizer, or `NULL` (default).
`invert`	a logical indicating whether to match separators between tokens.
`...`	further arguments to be passed to `gregexpr`.
`s`	a `String` object, or something coercible to this using `as.String` (e.g., a character string with appropriate encoding information)

Details

Regexp_Tokenizer() creates regexp tokenizers which use the given pattern and ... arguments to match tokens or separators between tokens via gregexpr, and then transform the results of this into character spans of the tokens found. The given description is currently kept as an attribute.

whitespace_tokenizer() tokenizes by treating any sequence of whitespace characters as a separator.

blankline_tokenizer() tokenizes by treating any sequence of blank lines as a separator.

wordpunct_tokenizer() tokenizes by matching sequences of alphabetic characters and sequences of (non-whitespace) non-alphabetic characters.

Value

Regexp_Tokenizer() returns the created regexp tokenizer.

blankline_tokenizer(), whitespace_tokenizer() and wordpunct_tokenizer() return the spans of the tokens found in s.

Examples

```
## A simple text.
s <- String("  First sentence.  Second sentence.  ")
##           ****5****0****5****0****5****0****5**

spans <- whitespace_tokenizer(s)
spans
s[spans]

spans <- wordpunct_tokenizer(s)
spans
s[spans]
```

Tree *Tree objects*

Description

Creation and manipulation of tree objects.

Usage

```
Tree(value, children = list())
## S3 method for class 'Tree'
format(x, width = 0.9 * getOption("width"), indent = 0,
       brackets = c("(", ")"), ...)
Tree_parse(x, brackets = c("(", ")"))
Tree_apply(x, f, recursive = FALSE)
```

Arguments

value	a (non-tree) node value of the tree.
children	a list giving the children of the tree.
x	a tree object for the format method and Tree_apply(); a character string for Tree_parse().
width	a positive integer giving the target column for a single-line nested bracketting.
indent	a non-negative integer giving the indentation used for formatting.
brackets	a character vector of length two giving the pair of opening and closing brackets to be employed for formatting or parsing.
...	further arguments passed to or from other methods.
f	a function to be applied to the children nodes.
recursive	a logical indicating whether to apply f recursively to the children of the children and so forth.

Details

Trees give hierarchical groupings of leaves and subtrees, starting from the root node of the tree. In natural language processing, the syntactic structure of sentences is typically represented by parse trees (e.g., http://en.wikipedia.org/wiki/Concrete_syntax_tree) and displayed using nested brackettings.

The tree objects in package **NLP** are patterned after the ones in NLTK (http://nltk.org), and primarily designed for representing parse trees. A tree object consists of the value of the root node and its children as a list of leaves and subtrees, where the leaves are elements with arbitrary non-tree values (and not subtrees with no children). The value and children can be extracted via $ subscripting using names value and children, respectively.

There is a format method for tree objects: this first tries a nested bracketting in a single line of the given width, and if this is not possible, produces a nested indented bracketting. The print method uses the format method, and hence its arguments to control the formatting.

Tree_parse() reads nested brackettings into a tree object.

Examples

```
x <- Tree(1, list(2, Tree(3, list(4)), 5))
format(x)
x$value
x$children

p <- Tree("VP",
          list(Tree("V",
                    list("saw")),
               Tree("NP",
                    list("him"))))
p <- Tree("S",
          list(Tree("NP",
                    list("I")),
               p))
```

```
p
## Force nested indented bracketting:
print(p, width = 10)

s <- "(S (NP I) (VP (V saw) (NP him)))"
p <- Tree_parse(s)
p

## Extract the leaves by recursively traversing the children and
## recording the non-tree ones:
Tree_leaf_gatherer <-
function()
{
    v <- list()
    list(update =
         function(e) if(!inherits(e, "Tree")) v <<- c(v, list(e)),
         value = function() v,
         reset = function() { v <<- list() })
}
g <- Tree_leaf_gatherer()
y <- Tree_apply(p, g$update, recursive = TRUE)
g$value()
```

`utils` *Annotation Utilities*

Description

Utilities for creating annotation objects.

Usage

```
next_id(id)
single_feature(value, tag)
```

Arguments

`id`	an integer vector of annotation ids.
`value`	an R object.
`tag`	a character string.

Details

`next_id()` obtains the next "available" id based on the given annotation ids (one more than the maximal non-missing id).

`single_feature()` creates a single feature from the given value and tag (i.e., a named list with the value named by the tag).

`viewers` *Text Document Viewers*

Description

Provide suitable "views" of the text contained in text documents.

Usage

```
words(x, ...)
sents(x, ...)
paras(x, ...)
tagged_words(x, ...)
tagged_sents(x, ...)
tagged_paras(x, ...)
chunked_sents(x, ...)
parsed_sents(x, ...)
parsed_paras(x, ...)
```

Arguments

`x`	a text document object.
`...`	further arguments to be passed to or from methods.

Value

For `words()`, a character vector with the word tokens in the document.

For `sents()`, a list of character vectors with the word tokens in each sentence.

For `paras()`, a list of lists of character vectors with the word tokens in each sentence, grouped according to the paragraphs.

For `tagged_words()`, a character vector with the POS tagged word tokens in the document (i.e., the word tokens and their POS tags, separated by `/`).

For `tagged_sents()`, a list of character vectors with the POS tagged word tokens in each sentence.

For `tagged_paras()`, a list of lists of character vectors with the POS tagged word tokens in each sentence, grouped according to the paragraphs.

For `chunked_sents()`, a list of (flat) `Tree` objects giving the chunk trees for each sentence in the document.

For `parsed_sents()`, a list of `Tree` objects giving the parse trees for each sentence in the document.

For `parsed_paras()`, a list of lists of `Tree` objects giving the parse trees for each sentence in the document, grouped according to the paragraphs in the document.

See Also

`TextDocument` for basic information on the text document infrastructure employed by package **NLP**.

`WordListDocument` *Word List Text Documents*

Description

Create text documents from word lists.

Usage

```
WordListDocument(con, encoding = "unknown", meta = list())
```

Arguments

`con`	a connection object or a character string. See `readLines` for details.
`encoding`	encoding to be assumed for input strings. See `readLines` for details.
`meta`	a named or empty list of document metadata tag-value pairs.

Details

`WordListDocument()` uses `readLines()` to read collections of words from connections for which each line provides one word, with blank lines ignored, and returns a word list document object which inherits from classes `"WordListDocument"` and `"TextDocument"`.

The words can be extracted using the `words` and `as.character` methods for class `"WordListDocument"`.

Value

An object inheriting from `"WordListDocument"` and `"TextDocument"`.

See Also

`TextDocument` for basic information on the text document infrastructure employed by package **NLP**.

Index

∗Topic **utilities**
language, 16
[.Annotation *(*Annotation*)*, 5
[.Span *(*Span*)*, 17
[[.Annotation *(*Annotation*)*, 5
[[.Span *(*Span*)*, 17
$<-.Annotation *(*Annotation*)*, 5
$<-.Span *(*Span*)*, 17

annotate, 2, *4*
AnnotatedPlainTextDocument, 3, *22*
Annotation, *2–4*, 5, *8*
annotations
*(*AnnotatedPlainTextDocument*)*, 3
annotations_in_spans, 7
Annotator, *2*, 8
annotators, 10
as.Annotation, *18*
as.Annotation *(*Annotation*)*, 5
as.character, *4*, *13*, *20*, *22*, *27*
as.data.frame.Annotation *(*Annotation*)*, 5
as.data.frame.Span *(*Span*)*, 17
as.Date, *14*
as.list.Annotation *(*Annotation*)*, 5
as.list.Span *(*Span*)*, 17
as.POSIXct, *14*
as.POSIXlt, *14*
as.Span, *5*
as.Span *(*Span*)*, 17
as.String, *2*, *3*, *22*
as.String *(*String*)*, 18

blankline_tokenizer *(*tokenizers*)*, 22

c.Annotation *(*Annotation*)*, 5
c.Span *(*Span*)*, 17
chunked_sents, *4*, *13*
chunked_sents *(*viewers*)*, 26
CoNLLTextDocument, 13, *22*
content, *22*
content *(*generics*)*, 15
content<- *(*generics*)*, 15

datetime, 14
duplicated.Annotation *(*Annotation*)*, 5
duplicated.Span *(*Span*)*, 17

format.Annotation *(*Annotation*)*, 5
format.Span *(*Span*)*, 17
format.Tree *(*Tree*)*, 23

generics, 15
gregexpr, *22*, *23*

is.Annotation *(*Annotation*)*, 5
is.Span *(*Span*)*, 17
is.String *(*String*)*, 18

language, 16
length.Annotation *(*Annotation*)*, 5
length.Span *(*Span*)*, 17

merge.Annotation *(*Annotation*)*, 5
meta, *22*
meta *(*generics*)*, 15
meta<- *(*generics*)*, 15

names.Annotation *(*Annotation*)*, 5
names.Span *(*Span*)*, 17
next_id *(*utils*)*, 25

Ops.Span *(*Span*)*, 17

paras, *4*, *20*
paras *(*viewers*)*, 26
parse_IETF_language_tag *(*language*)*, 16
parse_ISO_8601_datetime *(*datetime*)*, 14
parsed_paras, *4*
parsed_paras *(*viewers*)*, 26
parsed_sents, *4*
parsed_sents *(*viewers*)*, 26

Penn_Treebank_POS_tags (tagsets), 21
print.Annotation (Annotation), 5
print.Span (Span), 17
print.Tree (Tree), 23

readLines, *20*, *27*
Regexp_Tokenizer (tokenizers), 22

scan, *13*
sents, *4*, *13*, *20*, *22*
sents (viewers), 26
Simple annotator generators, *9*
Simple annotator generators (annotators), 10
Simple_Chunk_Annotator (annotators), 10
Simple_Entity_Annotator (annotators), 10
Simple_Para_Token_Annotator (annotators), 10
Simple_POS_Tag_Annotator (annotators), 10
Simple_Sent_Token_Annotator (annotators), 10
Simple_Stem_Annotator (annotators), 10
Simple_Word_Token_Annotator (annotators), 10
single_feature (utils), 25
Span, *5*, *8*, 17, *19*
String, *2*, *3*, 18, *22*
subset.Annotation (Annotation), 5

tagged_paras, *4*, *20*
tagged_paras (viewers), 26
tagged_sents, *4*, *13*, *20*
tagged_sents (viewers), 26
tagged_words, *4*, *13*, *20*
tagged_words (viewers), 26
TaggedTextDocument, 20, *22*
tagsets, 21
TextDocument, *4*, *13*, *14*, *16*, *20*, 21, *26*, *27*
tokenizers, 22
toString, *19*
Tree, 23, *26*
Tree_apply (Tree), 23
Tree_parse (Tree), 23

unique.Annotation (Annotation), 5
unique.Span (Span), 17
utils, 25

viewers, 26
whitespace_tokenizer (tokenizers), 22
WordListDocument, *22*, 27
wordpunct_tokenizer (tokenizers), 22
words, *4*, *13*, *20*, *22*, *27*
words (viewers), 26