Package: flexo (via r-universe)

September 16, 2024

Type Package

Title Simple Tools for Lexing/Parsing Text Data

Version 0.2.7

Author mikefc

Maintainer mikefc <mikefc@coolbutuseless.com>

Description Simple tools for lexing/parsing text data.

URL https://coolbutuseless.github.io/package/flexo, https://github.com/coolbutuseless/flexo

BugReports https://github.com/coolbutuseless/flexo/issues

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.1.2

Imports stringi, R6

Suggests knitr, rmarkdown, testthat, glue

VignetteBuilder knitr

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

RemoteUrl https://github.com/coolbutuseless/flexo

RemoteRef HEAD

RemoteSha 47701f62cd360e880d8fbd7a23ebf523aa014848

Contents

create_stream *An environment object encapsulating a stream of tokens and functions for manipulating/interrogating these token.*

Description

This is very similar to the R6 Class `TokenStream`, but it has no dependencies

Usage

```
create_stream(named_values)
```

Arguments

`named_values`	named vector containing the tokens. Usually the output from `lex()`

lex *Break a string into labelled tokens based upon a set of patterns*

Description

Break a string into labelled tokens based upon a set of patterns

Usage

```
lex(text, regexes, verbose = FALSE, ...)
```

Arguments

`text`	a single character string
`regexes`	a named vector of regex strings. Each string represents a regex to match a token, and the name of the string is the label for the token. Each regex can contain an explicit captured group using the standard `()` brackets. If a regex doesn't not define a captured group then the entire regex will be captured. The regexes will be processed in order such that an early match takes precedence over any later match.
`verbose`	print more information about the matching process. default: FALSE
`...`	further arguments passed to `stringi::stri_match_all()`. e.g. `multiline = TRUE`

Value

a named character vector with the names representing the token type with the value being the element extracted by the corresponding regular expression.

Examples

```
lex("hello there 123.45", regexes=c(number=re$number, word="(\\w+)", whitespace="(\\s+)"))
```

`re` *Regexes to match common elements*

Description

Regexes to match common elements

Usage

```
re
```

Format

An object of class `list` of length 3.

`TokenStream` *An R6 class for manipulating/interrogating a stream of tokens.*

Description

An R6 class for manipulating/interrogating a stream of tokens.

An R6 class for manipulating/interrogating a stream of tokens.

Public fields

`named_values` the original tokens

`position` current stream position Initialise a stream

Methods

Public methods:

- `TokenStream$new()`
- `TokenStream$reset()`
- `TokenStream$assert_within_range()`
- `TokenStream$check_within_range()`
- `TokenStream$check_name_seq()`
- `TokenStream$assert_name_seq()`
- `TokenStream$check_name()`
- `TokenStream$assert_name()`

- TokenStream$check_value_seq()
- TokenStream$assert_value_seq()
- TokenStream$check_value()
- TokenStream$assert_value()
- TokenStream$advance()
- TokenStream$read()
- TokenStream$read_names()
- TokenStream$read_values()
- TokenStream$consume()
- TokenStream$end_of_stream()
- TokenStream$read_while()
- TokenStream$consume_while()
- TokenStream$read_until()
- TokenStream$consume_until()
- TokenStream$print()
- TokenStream$clone()

Method `new()`**:**

Usage:

```
TokenStream$new(named_values)
```

Arguments:

`named_values` named vector of values Reset stream to the given absolute position.

Method `reset()`**:**

Usage:

```
TokenStream$reset(position = 1L)
```

Arguments:

`position` absolute position in stream. Default: 1 i.e. the start Throw an error if a read is not within range

Method `assert_within_range()`**:**

Usage:

```
TokenStream$assert_within_range(start, n)
```

Arguments:

`start, n` start position and number of values to read Check if a read is not within range

Method `check_within_range()`**:**

Usage:

```
TokenStream$check_within_range(start, n)
```

Arguments:

`start, n` start position and number of values to read

Returns: logical TRUE if values are within range of data Check the next names match the name sequence specified

Method `check_name_seq()`**:**

Usage:

```
TokenStream$check_name_seq(name_seq)
```

Arguments:

`name_seq` Expected sequence of names Assert the next names match the name sequence specified

Method `assert_name_seq()`**:**

Usage:

```
TokenStream$assert_name_seq(name_seq)
```

Arguments:

`name_seq` Expected sequence of names Check the next name is one of the valid names specified

Method `check_name()`**:**

Usage:

```
TokenStream$check_name(valid_names)
```

Arguments:

`valid_names` Valid names Assert the next name is one of the valid names specified

Method `assert_name()`**:**

Usage:

```
TokenStream$assert_name(valid_names)
```

Arguments:

`valid_names` Valid names Check the next values match the value sequence specified

Method `check_value_seq()`**:**

Usage:

```
TokenStream$check_value_seq(value_seq)
```

Arguments:

`value_seq` Expected sequence of values Assert the next values match the value sequence specified

Method `assert_value_seq()`**:**

Usage:

```
TokenStream$assert_value_seq(value_seq)
```

Arguments:

`value_seq` Expected sequence of values Check the next value is one of the valid values specified

Method `check_value()`**:**

Usage:

```
TokenStream$check_value(valid_values)
```

Arguments:

`valid_values` Valid values Assert the next value is one of the valid values specified

Method `assert_value()`**:**

Usage:

```
TokenStream$assert_value(valid_values)
```

Arguments:

`valid_values` Valid values Advance the stream

Method `advance()`**:**

Usage:

```
TokenStream$advance(n)
```

Arguments:

`n` number of tokens by which to advance the stream. May be negative. New position must be within range of the data Read n named values from the given position
Returns values but does not advance stream position

Method `read()`**:**

Usage:

```
TokenStream$read(n, offset = 0)
```

Arguments:

`n` number of values to read

`offset` offset from given position

Returns: named values at this position Read n names from the given position
Returns values but does not advance stream position

Method `read_names()`**:**

Usage:

```
TokenStream$read_names(n, offset = 0)
```

Arguments:

`n` number of values to read

`offset` offset from given position

Returns: names at this position Read n values from the given position
Returns values but does not advance stream position

Method `read_values()`**:**

Usage:

```
TokenStream$read_values(n, offset = 0)
```

Arguments:

`n` number of values to read

`offset` offset from given position

Returns: values at this position Consume n tokens from the given position i.e. read and advance the stream
Returns values and advances stream position.

Method `consume()`**:**

Usage:

```
TokenStream$consume(n)
```

Arguments:

`n` number of values to read

Returns: values starting at this position

Method `end_of_stream()`**:** has end of stream been reached? Read tokens while some expression matches

Returns values but does not advance stream position

Usage:

```
TokenStream$end_of_stream()
```

Method `read_while()`**:**

Usage:

```
TokenStream$read_while(name = NULL, value = NULL, combine = "or")
```

Arguments:

`name, value` the boundary of the consumption. if both name and value are specified, then `combine` indicates how to logically define the combination

`combine` logical operator value values: and, or Consume tokens while some expression matches Returns values and advances stream position.

Method `consume_while()`**:**

Usage:

```
TokenStream$consume_while(name = NULL, value = NULL, combine = "or")
```

Arguments:

`name, value` the boundary of the consumption. if both name and value are specified, then `combine` indicates how to logically define the combination

`combine` logical operator value values: and, or Read until some expression matches Returns values but does not advance stream position

Method `read_until()`**:**

Usage:

```
TokenStream$read_until(
  name = NULL,
  value = NULL,
  combine = "or",
  inclusive = TRUE
)
```

Arguments:

`name, value` the boundary of the consumption. if both name and value are specified, then `combine` indicates how to logically define the combination

`combine` logical operator value values: and, or

`inclusive` should the end-point be included in the returned results? Default: TRUE. If FALSE, then the end-point is not returned, and the stream position is set to *before* this end-point
Consume until some expression matches
Returns values and advances stream position.

Method `consume_until()`**:**

Usage:

```
TokenStream$consume_until(
  name = NULL,
  value = NULL,
  combine = "or",
  inclusive = TRUE
)
```

Arguments:

`name, value` the boundary of the consumption. if both name and value are specified, then `combine` indicates how to logically define the combination

`combine` logical operator value values: and, or

`inclusive` should the end-point be included in the returned results? Default: TRUE. If FALSE, then the end-point is not returned, and the stream position is set to *before* this end-point
Print current state

Method `print()`**:**

Usage:

```
TokenStream$print(n = 5)
```

Arguments:

`n` number of elements to print

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Index