

Package: glue (via r-universe)

September 27, 2024

Title Interpreted String Literals

Version 1.7.0.9000

Description An implementation of interpreted string literals, inspired by Python's Literal String Interpolation
<<https://www.python.org/dev/peps/pep-0498/>> and Docstrings
<<https://www.python.org/dev/peps/pep-0257/>> and Julia's Triple-Quoted String Literals
<<https://docs.julialang.org/en/v1.3/manual/strings/#Triple-Quoted-String-Literals-1>>.

License MIT + file LICENSE

URL <https://glue.tidyverse.org/>, <https://github.com/tidyverse/glue>

BugReports <https://github.com/tidyverse/glue/issues>

Depends R (>= 3.6)

Imports methods

Suggests crayon, DBI (>= 1.2.0), dplyr, knitr, magrittr, rlang, rmarkdown, RSQLite, testthat (>= 3.2.0), vctrs (>= 0.3.0), waldo (>= 0.5.3), withr

VignetteBuilder knitr

ByteCompile true

Config/Needs/website bench, forcats, ggbeeswarm, ggplot2, R.utils, rprintf, tidyr, tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Repository <https://tidyverse.r-universe.dev>

RemoteUrl <https://github.com/tidyverse/glue>

RemoteRef HEAD

RemoteSha 6a2423e5a0f1409ff2514a2f738171277a602956

Contents

as_glue	2
glue	3
glue_col	5
glue_collapse	7
glue_safe	8
glue_sql	9
identity_transformer	13
quoting	14
trim	14
Index	16

as_glue	<i>Coerce object to glue</i>
---------	------------------------------

Description

A glue object is a character vector with S3 class "glue". The "glue" class implements a print method that shows the literal contents (rather than the string implementation) and a + method, so that you can concatenate with the addition operator.

Usage

```
as_glue(x, ...)
```

Arguments

- x object to be coerced.
- ... further arguments passed to methods.

Value

A character vector with S3 class "glue".

Examples

```
x <- as_glue(c("abc", "\\\"\\\\\\\\", "\\n"))
x

x <- 1
y <- 3
glue("x + y") + " = {x + y}"
```

glue*Format and interpolate a string*

Description

Expressions enclosed by braces will be evaluated as R code. Long strings are broken by line and concatenated together. Leading whitespace and blank lines from the first and last lines are automatically trimmed.

Usage

```
glue_data(  
  .x,  
  ...,  
  .sep = "",  
  .envir = parent.frame(),  
  .open = "{",  
  .close = "}",  
  .na = "NA",  
  .null = character(),  
  .comment = "#",  
  .literal = FALSE,  
  .transformer = identity_transformer,  
  .trim = TRUE  
)  
  
glue(  
  ...,  
  .sep = "",  
  .envir = parent.frame(),  
  .open = "{",  
  .close = "}",  
  .na = "NA",  
  .null = character(),  
  .comment = "#",  
  .literal = FALSE,  
  .transformer = identity_transformer,  
  .trim = TRUE  
)
```

Arguments

<code>.x</code>	[listish] An environment, list, or data frame used to lookup values.
<code>...</code>	[expressions] Unnamed arguments are taken to be expression string(s) to format. Multiple

inputs are concatenated together before formatting. Named arguments are taken to be temporary variables available for substitution.

For `glue_data()`, elements in `...` override the values in `.x`.

<code>.sep</code>	[character(1): ''] Separator used to separate elements.
<code>.envir</code>	[environment: parent.frame()] Environment to evaluate each expression in. Expressions are evaluated from left to right. If <code>.x</code> is an environment, the expressions are evaluated in that environment and <code>.envir</code> is ignored. If NULL is passed, it is equivalent to <code>emptyenv()</code> .
<code>.open</code>	[character(1): '{'] The opening delimiter. Doubling the full delimiter escapes it.
<code>.close</code>	[character(1): '}'] The closing delimiter. Doubling the full delimiter escapes it.
<code>.na</code>	[character(1): 'NA'] Value to replace NA values with. If NULL missing values are propagated, that is an NA result will cause NA output. Otherwise the value is replaced by the value of <code>.na</code> .
<code>.null</code>	[character(1): 'character()'] Value to replace NULL values with. If <code>character()</code> whole output is <code>character()</code> . If NULL all NULL values are dropped (as in <code>paste0()</code>). Otherwise the value is replaced by the value of <code>.null</code> .
<code>.comment</code>	[character(1): '#'] Value to use as the comment character.
<code>.literal</code>	[boolean(1): 'FALSE'] Whether to treat single or double quotes, backticks, and comments as regular characters (vs. as syntactic elements), when parsing the expression string. Setting <code>.literal = TRUE</code> probably only makes sense in combination with a custom <code>.transformer</code> , as is the case with <code>glue_col()</code> . Regard this argument (especially, its name) as experimental.
<code>.transformer</code>	[function] A function taking two arguments, <code>text</code> and <code>envir</code> , where <code>text</code> is the unparsed string inside the glue block and <code>envir</code> is the execution environment. A <code>.transformer</code> lets you modify a glue block before, during, or after evaluation, allowing you to create your own custom <code>glue()</code> -like functions. See <code>vignette("transformers")</code> for examples.
<code>.trim</code>	[logical(1): 'TRUE'] Whether to trim the input template with <code>trim()</code> or not.

Value

A glue object, as created by `as_glue()`.

See Also

<https://www.python.org/dev/peps/pep-0498/> and <https://www.python.org/dev/peps/pep-0257/> upon which this is based.

Examples

```

name <- "Fred"
age <- 50
anniversary <- as.Date("1991-10-12")
glue('My name is {name}, ',
      'my age next year is {age + 1}, ',
      'my anniversary is {format(anniversary, "%A, %B %d, %Y")}.')

# single braces can be inserted by doubling them
glue("My name is {name}, not {{name}}.")

# Named arguments can be used to assign temporary variables.
glue('My name is {name}, ',
      ' my age next year is {age + 1}, ',
      ' my anniversary is {format(anniversary, "%A, %B %d, %Y")}.',
      name = "Joe",
      age = 40,
      anniversary = as.Date("2001-10-12"))

# `glue()` can also be used in user defined functions
intro <- function(name, profession, country){
  glue("My name is {name}, a {profession}, from {country}")
}
intro("Shelmith", "Senior Data Analyst", "Kenya")
intro("Cate", "Data Scientist", "Kenya")

# `glue_data()` is useful in magrittr pipes
if (require(magrittr)) {

  mtcars %>% glue_data("{rownames(.)} has {hp} hp")

  # Or within dplyr pipelines
  if (require(dplyr)) {

    head(iris) %>%
      mutate(description = glue("This {Species} has a petal length of {Petal.Length}"))

  }}

# Alternative delimiters can also be used if needed
one <- "1"
glue("The value of $e^{2\\pi i}$ is $<<one>>$.", .open = "<<", .close = ">>")

```

glue_col

Construct strings with color

Description

The [crayon](#) package defines a number of functions used to color terminal output. `glue_col()` and `glue_data_col()` functions provide additional syntax to make using these functions in glue strings easier.

Using the following syntax will apply the function `crayon::blue()` to the text 'foo bar'.

```
{blue foo bar}
```

If you want an expression to be evaluated, simply place that in a normal brace expression (these can be nested).

```
{blue 1 + 1 = {1 + 1}}
```

If the text you want to color contains, e.g., an unpaired quote or a comment character, specify `.literal = TRUE`.

Usage

```
glue_col(..., .envir = parent.frame(), .na = "NA", .literal = FALSE)
```

```
glue_data_col(.x, ..., .envir = parent.frame(), .na = "NA", .literal = FALSE)
```

Arguments

<code>...</code>	<p>[expressions] Unnamed arguments are taken to be expression string(s) to format. Multiple inputs are concatenated together before formatting. Named arguments are taken to be temporary variables available for substitution. For <code>glue_data()</code>, elements in <code>...</code> override the values in <code>.x</code>.</p>
<code>.envir</code>	<p>[environment: <code>parent.frame()</code>] Environment to evaluate each expression in. Expressions are evaluated from left to right. If <code>.x</code> is an environment, the expressions are evaluated in that environment and <code>.envir</code> is ignored. If <code>NULL</code> is passed, it is equivalent to <code>emptyenv()</code>.</p>
<code>.na</code>	<p>[character(1): 'NA'] Value to replace NA values with. If <code>NULL</code> missing values are propagated, that is an NA result will cause NA output. Otherwise the value is replaced by the value of <code>.na</code>.</p>
<code>.literal</code>	<p>[boolean(1): 'FALSE'] Whether to treat single or double quotes, backticks, and comments as regular characters (vs. as syntactic elements), when parsing the expression string. Setting <code>.literal = TRUE</code> probably only makes sense in combination with a custom <code>.transformer</code>, as is the case with <code>glue_col()</code>. Regard this argument (especially, its name) as experimental.</p>
<code>.x</code>	<p>[listish] An environment, list, or data frame used to lookup values.</p>

Value

A glue object, as created by `as_glue()`.

Examples

```

library(crayon)

glue_col("{blue foo bar}")

glue_col("{blue 1 + 1 = {1 + 1}}")

glue_col("{blue 2 + 2 = {green {2 + 2}}}")

white_on_black <- bgBlack $ white
glue_col("{white_on_black
  Roses are {red {colors()[[552]]}},
  Violets are {blue {colors()[[26]]}},
  `glue_col()` can show \\
  {red c}{yellow o}{green l}{cyan o}{blue r}{magenta s}
  and {bold bold} and {underline underline} too!
}")

# this would error due to an unterminated quote, if we did not specify
# `.literal = TRUE`
glue_col("{yellow It's} happening!", .literal = TRUE)

# `.literal = TRUE` also prevents an error here due to the `#` comment
glue_col(
  "A URL: {magenta https://github.com/tidyverse/glue#readme}",
  .literal = TRUE
)

# `.literal = TRUE` does NOT prevent evaluation
x <- "world"
y <- "day"
glue_col("hello {x}! {green it's a new {y}!}", .literal = TRUE)

```

glue_collapse

*Collapse a character vector***Description**

`glue_collapse()` collapses a character vector of any length into a length 1 vector. `glue_sql_collapse()` does the same but returns a `[DBI::SQL()]` object rather than a glue object.

Usage

```

glue_collapse(x, sep = "", width = Inf, last = "")

glue_sql_collapse(x, sep = "", width = Inf, last = "")

```

Arguments

<code>x</code>	The character vector to collapse.
<code>sep</code>	a character string to separate the terms. Not <code>NA_character_</code> .
<code>width</code>	The maximum string width before truncating with <code>...</code>
<code>last</code>	String used to separate the last two items if <code>x</code> has at least 2 items.

Value

Always returns a length-1 glue object, as created by `as_glue()`.

Examples

```
glue_collapse(glue("{1:10}"))

# Wide values can be truncated
glue_collapse(glue("{1:10}"), width = 5)

glue_collapse(1:4, ", ", last = " and ")
```

<code>glue_safe</code>	<i>Safely interpolate strings</i>
------------------------	-----------------------------------

Description

`glue_safe()` and `glue_data_safe()` differ from `glue()` and `glue_data()` in that the safe versions only look up symbols from an environment using `get()`. They do not execute any R code. This makes them suitable for use with untrusted input, such as inputs in a Shiny application, where using the normal functions would allow an attacker to execute arbitrary code.

Usage

```
glue_safe(..., .envir = parent.frame())

glue_data_safe(.x, ..., .envir = parent.frame())
```

Arguments

<code>...</code>	[expressions] Unnamed arguments are taken to be expression string(s) to format. Multiple inputs are concatenated together before formatting. Named arguments are taken to be temporary variables available for substitution. For <code>glue_data()</code> , elements in <code>...</code> override the values in <code>.x</code> .
<code>.envir</code>	[environment: <code>parent.frame()</code>] Environment to evaluate each expression in. Expressions are evaluated from left to right. If <code>.x</code> is an environment, the expressions are evaluated in that environment and <code>.envir</code> is ignored. If <code>NULL</code> is passed, it is equivalent to <code>emptyenv()</code> .
<code>.x</code>	[listish] An environment, list, or data frame used to lookup values.

Value

A glue object, as created by `as_glue()`.

Examples

```
"1 + 1" <- 5
# glue actually executes the code
glue("{1 + 1}")

# glue_safe just looks up the value
glue_safe("{1 + 1}")

rm("1 + 1")
```

glue_sql

*Interpolate strings with SQL escaping***Description**

SQL databases often have custom quotation syntax for identifiers and strings which make writing SQL queries error prone and cumbersome to do. `glue_sql()` and `glue_data_sql()` are analogs to `glue()` and `glue_data()` which handle the SQL quoting. `glue_sql_collapse()` can be used to collapse `DBI::SQL()` objects.

They automatically quote character results, quote identifiers if the glue expression is surrounded by backticks `` and do not quote non-characters such as numbers. If numeric data is stored in a character column (which should be quoted) pass the data to `glue_sql()` as a character.

Returning the result with `DBI::SQL()` will suppress quoting if desired for a given value.

Note **parameterized queries** are generally the safest and most efficient way to pass user defined values in a query, however not every database driver supports them.

If you place a `*` at the end of a glue expression the values will be collapsed with commas, or if there are no values, produce NULL. This is useful for (e.g.) the **SQL IN Operator**.

Usage

```
glue_sql(
  ...,
  .con,
  .sep = "",
  .envir = parent.frame(),
  .open = "{",
  .close = "}",
  .na = DBI::SQL("NULL"),
  .null = character(),
  .comment = "#",
  .literal = FALSE,
  .trim = TRUE
```

```

)

glue_data_sql(
  .x,
  ...,
  .con,
  .sep = "",
  .envir = parent.frame(),
  .open = "{",
  .close = "}",
  .na = DBI::SQL("NULL"),
  .null = character(),
  .comment = "#",
  .literal = FALSE,
  .trim = TRUE
)

```

Arguments

...	<p>[expressions]</p> <p>Unnamed arguments are taken to be expression string(s) to format. Multiple inputs are concatenated together before formatting. Named arguments are taken to be temporary variables available for substitution.</p> <p>For <code>glue_data()</code>, elements in ... override the values in <code>.x</code>.</p>
.con	[DBIConnection]: A DBI connection object obtained from <code>DBI::dbConnect()</code> .
.sep	<p>[character(1): ""]</p> <p>Separator used to separate elements.</p>
.envir	<p>[environment: parent.frame()]</p> <p>Environment to evaluate each expression in. Expressions are evaluated from left to right. If <code>.x</code> is an environment, the expressions are evaluated in that environment and <code>.envir</code> is ignored. If <code>NULL</code> is passed, it is equivalent to <code>emptyenv()</code>.</p>
.open	<p>[character(1): '{']</p> <p>The opening delimiter. Doubling the full delimiter escapes it.</p>
.close	<p>[character(1): '}']</p> <p>The closing delimiter. Doubling the full delimiter escapes it.</p>
.na	<p>[character(1): DBI::SQL("NULL")]</p> <p>Value to replace NA values with. If <code>NULL</code> missing values are propagated, that is an NA result will cause NA output. Otherwise the value is replaced by the value of <code>.na</code>.</p>
.null	<p>[character(1): 'character()']</p> <p>Value to replace <code>NULL</code> values with. If <code>character()</code> whole output is <code>character()</code>. If <code>NULL</code> all <code>NULL</code> values are dropped (as in <code>paste0()</code>). Otherwise the value is replaced by the value of <code>.null</code>.</p>
.comment	<p>[character(1): '#']</p> <p>Value to use as the comment character.</p>

<code>.literal</code>	<code>[boolean(1): 'FALSE']</code> Whether to treat single or double quotes, backticks, and comments as regular characters (vs. as syntactic elements), when parsing the expression string. Setting <code>.literal = TRUE</code> probably only makes sense in combination with a custom <code>.transformer</code> , as is the case with <code>glue_col()</code> . Regard this argument (especially, its name) as experimental.
<code>.trim</code>	<code>[logical(1): 'TRUE']</code> Whether to trim the input template with <code>trim()</code> or not.
<code>.x</code>	<code>[listish]</code> An environment, list, or data frame used to lookup values.

Value

A `DBI::SQL()` object with the given query.

See Also

`glue_sql_collapse()` to collapse `DBI::SQL()` objects.

Examples

```
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
iris2 <- iris
colnames(iris2) <- gsub("[.]", "_", tolower(colnames(iris)))
DBI::dbWriteTable(con, "iris", iris2)
var <- "sepal_width"
tbl <- "iris"
num <- 2
val <- "setosa"
glue_sql("
  SELECT {`var`}
  FROM {`tbl`}
  WHERE {`tbl`}.sepal_length > {num}
    AND {`tbl`}.species = {val}
", .con = con)

# If sepal_length is store on the database as a character explicitly convert
# the data to character to quote appropriately.
glue_sql("
  SELECT {`var`}
  FROM {`tbl`}
  WHERE {`tbl`}.sepal_length > {as.character(num)}
    AND {`tbl`}.species = {val}
", .con = con)

# `glue_sql()` can be used in conjunction with parameterized queries using
# `DBI::dbBind()` to provide protection for SQL Injection attacks
sql <- glue_sql("
  SELECT {`var`}
  FROM {`tbl`}
```

```

      WHERE {`tbl`}.sepal_length > ?
    ", .con = con)
query <- DBI::dbSendQuery(con, sql)
DBI::dbBind(query, list(num))
DBI::dbFetch(query, n = 4)
DBI::dbClearResult(query)

# `glue_sql()` can be used to build up more complex queries with
# interchangeable sub queries. It returns `DBI::SQL()` objects which are
# properly protected from quoting.
sub_query <- glue_sql("
  SELECT *
  FROM {`tbl`}
  ", .con = con)

glue_sql("
  SELECT s.{`var`}
  FROM ({sub_query}) AS s
  ", .con = con)

# If you want to input multiple values for use in SQL IN statements put `*`
# at the end of the value and the values will be collapsed and quoted appropriately.
glue_sql("SELECT * FROM {`tbl`} WHERE sepal_length IN ({vals*})",
  vals = 1, .con = con)

glue_sql("SELECT * FROM {`tbl`} WHERE sepal_length IN ({vals*})",
  vals = 1:5, .con = con)

glue_sql("SELECT * FROM {`tbl`} WHERE species IN ({vals*})",
  vals = "setosa", .con = con)

glue_sql("SELECT * FROM {`tbl`} WHERE species IN ({vals*})",
  vals = c("setosa", "versicolor"), .con = con)

# If you need to reference variables from multiple tables use `DBI::Id()`.
# Here we create a new table of nicknames, join the two tables together and
# select columns from both tables. Using `DBI::Id()` and the special
# `glue_sql()` syntax ensures all the table and column identifiers are quoted
# appropriately.

iris_db <- "iris"
nicknames_db <- "nicknames"

nicknames <- data.frame(
  species = c("setosa", "versicolor", "virginica"),
  nickname = c("Beachhead Iris", "Harlequin Blueflag", "Virginia Iris"),
  stringsAsFactors = FALSE
)

DBI::dbWriteTable(con, nicknames_db, nicknames)

cols <- list(
  DBI::Id(iris_db, "sepal_length"),

```

```
  DBI::Id(iris_db, "sepal_width"),
  DBI::Id(nicknames_db, "nickname")
)

iris_species <- DBI::Id(iris_db, "species")
nicknames_species <- DBI::Id(nicknames_db, "species")

query <- glue_sql("
  SELECT `{cols}`*
  FROM `{iris_db}`
  JOIN `{nicknames_db}`
  ON `{iris_species}`={`nicknames_species`}",
  .con = con
)
query

DBI::dbGetQuery(con, query, n = 5)

DBI::dbDisconnect(con)
```

identity_transformer *Parse and Evaluate R code*

Description

This is a simple wrapper around `eval(parse())`, used as the default transformer.

Usage

```
identity_transformer(text, envir = parent.frame())
```

Arguments

<code>text</code>	Text (typically) R code to parse and evaluate.
<code>envir</code>	environment to evaluate the code in

See Also

`vignette("transformers", "glue")` for documentation on creating custom glue transformers and some common use cases.

quoting

*Quoting operators***Description**

These functions make it easy to quote each individual element and are useful in conjunction with `glue_collapse()`. These are thin wrappers around `base::encodeString()`.

Usage

```
single_quote(x)
```

```
double_quote(x)
```

```
backtick(x)
```

Arguments

`x` A character to quote.

Value

A character vector of the same length as `x`, with the same attributes (including names and dimensions) but with no class set.

Marked UTF-8 encodings are preserved.

Examples

```
x <- 1:5
glue('Values of x: {glue_collapse(backtick(x), sep = ", ", last = " and ")}')
```

trim

*Trim a character vector***Description**

This trims a character vector according to the trimming rules used by `glue`. These follow similar rules to [Python Docstrings](#), with the following features.

- Leading and trailing whitespace from the first and last lines is removed.
- A uniform amount of indentation is stripped from the second line on, equal to the minimum indentation of all non-blank lines after the first.
- Lines can be continued across newlines by using `\\`.

Usage

```
trim(x)
```

Arguments

x A character vector to trim.

Value

A character vector.

Examples

```
glue("
  A formatted string
  Can have multiple lines
  with additional indention preserved
")
```

```
glue("
  \ntrailing or leading newlines can be added explicitly\n
")
```

```
glue("
  A formatted string \\
  can also be on a \\
  single line
")
```

Index

as_glue, [2](#)
as_glue(), [4](#), [6](#), [8](#), [9](#)

backtick (quoting), [14](#)
base::encodeString(), [14](#)

crayon, [5](#)
crayon::blue(), [6](#)

DBI::dbConnect(), [10](#)
DBI::SQL(), [9](#), [11](#)
double_quote (quoting), [14](#)

emptyenv(), [4](#), [6](#), [8](#), [10](#)

get(), [8](#)
glue, [3](#)
glue(), [8](#), [9](#)
glue_col, [5](#)
glue_collapse, [7](#)
glue_collapse(), [14](#)
glue_data (glue), [3](#)
glue_data(), [8](#), [9](#)
glue_data_col (glue_col), [5](#)
glue_data_safe (glue_safe), [8](#)
glue_data_sql (glue_sql), [9](#)
glue_safe, [8](#)
glue_sql, [9](#)
glue_sql_collapse (glue_collapse), [7](#)
glue_sql_collapse(), [11](#)

identity_transformer, [13](#)

NA_character_, [8](#)

quoting, [14](#)

single_quote (quoting), [14](#)

trim, [14](#)
trim(), [4](#), [11](#)