General description of transformers functionality in {glue} with some potentially useful examples.

**Prologue**

Package {[glue](https://glue.tidyverse.org/index.html)} is designed as "small, fast, dependency free�? tools to "glue strings to data in R�?. To put simply, it provides concise and flexible alternatives for paste() with some additional features:

library(glue)

x <- 10

paste("I have", x, "apples.")

## [1] "I have 10 apples."

glue("I have {x} apples.")

## I have 10 apples.

Recently, fate lead me to try using {glue} in a package. I was very pleased to how it makes code more readable, which I believe is a very important during package development. However, I stumbled upon this pretty unexpected behavior:

y <- NULL

paste("I have", x, "apples and", y, "oranges.")

## [1] "I have 10 apples and oranges."

str(glue("I have {x} apples and {y} oranges."))

## Classes 'glue', 'character' chr(0)

If one of the expressions is evaluated into NULL then the output becomes empty string. This was unintuitive result and for a while I thought about stop using {glue} because NULL is expected to be a valid input.

Basically, transformer is a function that changes the output of R expressions the way you want. As I wanted to make NULL visible, this is a perfect way to do it.

**Overview**

This post describes an easy way to create {glue} wrappers with custom transformers. It also lists some examples that can be helpful in common tasks:

* **Transformers** uses a little bit of functional programming magic to create a potentially useful transformers.

Setup is very simple this time:

# {glue} was loaded in examples

# For functional programming magic

library(purrr)

# For string manipulation in one of the examples

library(stringr)

**Transformers**

The task of creating wrapper for glue() essentially consists from two parts:

* Evaluate properly a supplied R expressions.
* Modify them to show intended behavior.

The transforming\_glue() wrapper does exactly this:

transforming\_glue <- function(transformer) {

function(..., .sep = "", .envir = parent.frame(), .open = "{", .close = "}",

.na = "NA") {

glue(

..., .sep = .sep, .envir = .envir, .open = .open, .close = .close,

.na = "NA",

.transformer = compose(transformer, identity\_transformer)

)

}

}

Breakdown of this code:

* Input is a transformer – function that takes an already evaluated R object and modifies it the way you want.
* Output is a function that is a wrapper for glue(). Its transformer is a function composition that first evaluates R expression with identity\_transformer (function from {glue}) and then applies supplied transformer. Composition here is done with [compose()](https://purrr.tidyverse.org/reference/compose.html) – an element of functional programming magic from {[purrr](https://purrr.tidyverse.org/" \t "_blank)}.

**Show NULL**

Back to initial problem. We want NULL to be a valid R value for a glue():

show\_null <- function(x, val = "NULL") {

if (is.null(x)) {

val

} else {

x

}

}

glue\_null <- transforming\_glue(show\_null)

# Example from Prologue

glue\_null("I have {x} apples and {y} oranges.")

## I have 10 apples and NULL oranges.

**Fixed width output**

With {[stringr](https://stringr.tidyverse.org/" \t "_blank)} package you can force an output to be fixed width:

str\_width <- function(x, width) {

if (str\_length(x) > width) {

str\_trunc(x, width, side = "right")

} else {

str\_pad(x, width, side = "right")

}

}

glue\_width <- transforming\_glue(partial(str\_width, width = 10))

short\_oh <- "Ooh!"

long\_oh <- "Oooooooooooh!"

glue\_width("This puzzles ({short\_oh}) and surprises ({long\_oh}) me.")

## This puzzles (Ooh! ) and surprises (Ooooooo...) me.

**Note** usage of [partial()](https://purrr.tidyverse.org/reference/partial.html) here: it takes function along with its arguments' values and modifies it by "pre-filling�? those arguments.

**Enclose output**

In some situation you might want to explicitly show which strings represent R objects in the output. You can do that by enclosing the output in some sort of braces:

enclose <- function(x, start = "<", end = ">") {

paste0(start, x, end)

}

glue\_enclose <- transforming\_glue(enclose)

glue\_enclose("What if I had {x} oranges?")

## What if I had <10> oranges?

**Bizarro encryption**

One possibly useful pattern is to encrypt the used data to prevent it from seeing by untrustworthy eyes. Here we will use simplified bizarro(). Here glue\_bizarro() "reverts�? R objects based on their type.

str\_reverse <- function(x) {

vapply(

strsplit(x, ""),

FUN = function(z) paste(rev(z), collapse = ""),

FUN.VALUE = ""

)

}

bizarro <- function(x) {

cls <- class(x)[[1]]

switch(

cls,

logical = !x,

integer = -x,

numeric = -x,

character = str\_reverse(x),

x

)

}

glue\_bizarro <- transforming\_glue(bizarro)

new\_fruit <- "pomegranate"

glue\_bizarro(

"Then I might have {x + 10} apples. Is that {TRUE}?

Maybe I want {new\_fruit}?"

)

## Then I might have -20 apples. Is that FALSE?

## Maybe I want etanargemop?

**Ultimate example**

Using already familiar functional programming technique, we can create an ultimate glue() wrapper as a combination, or rather compose()-ition, of all previous examples. The most important part is supply them in correct order:

glue\_ultimate <- transforming\_glue(

compose(

enclose,

partial(str\_width, width = 10),

# To ensure that input of `str\_width()` is character

as.character,

show\_null,

bizarro

)

)

glue\_ultimate(

"I have {x} apples and {y} oranges.

This puzzles ({short\_oh}) and surprises ({long\_oh}) me.

What if I had {x} oranges?

Then I might have {x + 10} apples. Is that {TRUE}?

Maybe I want {new\_fruit}?"

)

## I have <-10 > apples and oranges.

## This puzzles () and surprises () me.

## What if I had <-10 > oranges?

## Then I might have <-20 > apples. Is that ?

## Maybe I want ?

**Conclusions**

* Package {glue} is a very useful and flexible way of creating strings based on evaluation of R expressions.
* Its "transformer�? functionality is an interesting way to manipulate string output by supplying custom modification function.
* Functional programming with {purrr} can be very helpful in creating concise and extensible code.

sessionInfo()

sessionInfo()

## R version 3.4.4 (2018-03-15)

## Platform: x86\_64-pc-linux-gnu (64-bit)

## Running under: Ubuntu 16.04.5 LTS

##

## Matrix products: default

## BLAS: /usr/lib/openblas-base/libblas.so.3

## LAPACK: /usr/lib/libopenblasp-r0.2.18.so

##

## locale:

## [1] LC\_CTYPE=ru\_UA.UTF-8 LC\_NUMERIC=C

## [3] LC\_TIME=ru\_UA.UTF-8 LC\_COLLATE=ru\_UA.UTF-8

## [5] LC\_MONETARY=ru\_UA.UTF-8 LC\_MESSAGES=ru\_UA.UTF-8

## [7] LC\_PAPER=ru\_UA.UTF-8 LC\_NAME=C

## [9] LC\_ADDRESS=C LC\_TELEPHONE=C

## [11] LC\_MEASUREMENT=ru\_UA.UTF-8 LC\_IDENTIFICATION=C

##

## attached base packages:

## [1] methods stats graphics grDevices utils datasets base

##

## other attached packages:

## [1] stringr\_1.3.1 purrr\_0.2.5 glue\_1.3.0

##

## loaded via a namespace (and not attached):

## [1] Rcpp\_0.12.18 bookdown\_0.7 crayon\_1.3.4 digest\_0.6.15

## [5] rprojroot\_1.3-2 backports\_1.1.2 magrittr\_1.5 evaluate\_0.11

## [9] blogdown\_0.8 rlang\_0.2.1.9000 stringi\_1.2.4 rmarkdown\_1.10

## [13] tools\_3.4.4 xfun\_0.3 yaml\_2.2.0 compiler\_3.4.4

## [17] htmltools\_0.3.6 knitr\_1.20