Simple variable interpolation in R

Zuguang Gu <z.gu@dkfz.de>

March 25, 2016

There are several ways to construct strings in *R* such as `paste`. However, when the string which is going to be constructed is so complicated, using `paste()` will be a nightmare. For example, we want to put some parameters as title in a plot.

```
region = c(1, 2)
value = 4
name = "name"
str = paste("region = (", region[1], ", ", region[2], "), value = ", value,
            ", name = '", name, "'", sep = "")
cat(str)

## region = (1, 2), value = 4, name = 'name'
```

As you can see, it is hard to read and very easy to make mistakes. (Syntax highlighting may be helpful to match quotes, but it is still quite annoying to see so many quotes.)

In *Perl*, we always use variable interpolation to construct complicated strings in which variables are started with special marks (sigil), and variables will be replaced with their real values. In this package, we aim to implement variable interpolation in *R*. The idea is rather simple: use special marks to identify variables and then replace with their values. The function here is `qq()` which is named from the subroutine with the same name in *Perl* (It stands for double quote). Using variable interpolation, above example can be written as:

```
library(GetoptLong)

str = qq("region = (@{region[1]}, @{region[2]}), value = @{value}, name = '@{name}'")
cat(str)

## region = (1, 2), value = 4, name = 'name'
```

Or use the shortcut function `qqcat()`:

```
qqcat("region = (@{region[1]}, @{region[2]}), value = @{value}, name = '@{name}'")

## region = (1, 2), value = 4, name = 'name'
```

One feature of `qqcat()` is you can set a global prefix to the messages by `qq.options("cat_prefix")`, either a string or a function. If it is set as a function, the value will be generated at real time by executing the function.

```
qq.options("cat_prefix" = "[INFO] ")
qqcat("This is a message")

## [INFO] This is a message

qq.options("cat_prefix" = function() format(Sys.time(), "[%Y-%m-%d %H:%M:%S] "))
qqcat("This is a message")

## [2016-03-25 18:12:58] This is a message
```

```
Sys.sleep(2)
qqcat("This is a message after 2 seconds")

## [2016-03-25 18:13:00] This is a message after 2 seconds

qq.options("cat_prefix" = "")
qqcat("This is a message")

## This is a message
```

You can shut down all messages produced by `qqcat()` by `qq.options("cat_verbose" = FALSE)`.

```
qq.options("cat_prefix" = "[INFO] ", "cat_verbose" = FALSE)
qqcat("This is a message")
```

Also you can set a prefix which has local effect.

```
qq.options(RESET = TRUE)
qq.options("cat_prefix" = "[DEBUG] ")
qqcat("This is a message", cat_prefix = "[INFO] ")

## [INFO] This is a message

qqcat("This is a message")

## [DEBUG] This is a message
```

From version 1.1.2, `qq.options()` can work in a local mode in which the copy of the options only work in a local chunk.

```
qq.options(LOCAL = TRUE)
qq.options("cat_prefix" = "[INFO] ")
qqcat("This is the first message")

## [INFO] This is the first message

qqcat("This is the second message")

## [INFO] This is the second message

qq.options(LOCAL = FALSE)
qqcat("This is the third message")

## [DEBUG] This is the third message
```

Reset the options so that it does not affect example code in following part of the vignette.

```
qq.options(RESET = TRUE)
```

You can use `cat_verbose` together with `GetoptLong()` when you want to run the R script in command line. In following example code, if you do not specify `--verbose`, all messages will be shut down.

```
library(GetoptLong)
GetoptLong(c(
    "verbose", "Print message"
))
qq.options("cat_verbose" = verbose)
qqcat("This is a message")
```

```
qq.options(RESET = TRUE)
```

Not only simple scalars but also pieces of codes can be interpolated:

```
n = 1
qqcat("There @{ifelse(n == 1, 'is', 'are')} @{n} dog@{ifelse(n == 1, '', 's')}.\n")

## There is 1 dog.

n = 2
qqcat("There @{ifelse(n == 1, 'is', 'are')} @{n} dog@{ifelse(n == 1, '', 's')}.\n")

## There are 2 dogs.
```

NOTE: Since `qq` as the function name is very easy to be used by other packages (E.g., in **lattice**, there is also a `qq()` function as well) and if so, you may enforce `qq()` in your working environment as the function in **GetoptLong** by:

```
qq = GetoptLong::qq
```

1 Code patterns

In above exmaple, `@{}` is used to mark variables. Later, variable names will be extracted from these marks and replaced with their real values.

The marking code pattern can be any type. But you should make sure it is easy to tell the difference from other part in the string. You can set your code pattern as an argument in `qq()`. The default pattern is `@\\{CODE\\}` because we only permit `CODE` to return simple vectors and `@` is a sigil representing array in *Perl*.

```
x = 1
qqcat("x = #{x}", code.pattern = "#\\{CODE\\}")

## x = 1
```

Or set in `qq.options()` as a global setting:

```
qq.options("code.pattern" = "#\\{CODE\\}")
```

As you can guess, in `@\\{CODE\\}`, `CODE` will be replaced with `.*?` to construct a regular expression and to match variable names in the string. So if your `code.pattern` contains special characters, make sure to escape them. Some candidate `code.pattern` are:

```
code.pattern = "@\\{CODE\\}"    # default style
code.pattern = "@\\[CODE\\]"
code.pattern = "@\\(CODE\\)"
code.pattern = "%\\{CODE\\}"
code.pattern = "%\\[CODE\\]"
code.pattern = "%\\(CODE\\)"
code.pattern = "\\$\\{CODE\\}"
code.pattern = "\\$\\[CODE\\]"
code.pattern = "\\$\\(CODE\\)"
code.pattern = "#\\{CODE\\}"
code.pattern = "#\\[CODE\\]"
code.pattern = "#\\(CODE\\)"
code.pattern = "\\[%CODE%\\]"  # Template Toolkit (Perl module) style :)
```

Since we just replace `CODE` to `.*?`, the function will only match to the first right parentheses/brackets. (In *Perl*, I always use recursive regular expression to extract such pairing parentheses. But in *R*, it seems difficult.) So, for example, if you are using `@\\[CODE\\]` and your string is `"@[a[1]]"`, it will fail to extract the correct variable name while only extracts `a[1`, finally it generates an error when executing `a[1`. In such condition, you should use other pattern styles that do not contain `[]`.

Finally, I suggest a more safe code pattern style that you do not need to worry about parentheses stuff:

```
code.pattern = "`CODE`"
```

2 Where to look for variables

It will first look up in the envoking environment, then through searching path. Users can also pass values of variables as a list like:

```
x = 1
y = 2
qqcat("x = @{x}, y = @{y}", envir = list(x = "a", y = "b"))

## x = a, y = b
```

If variables are passed through list, `qq()` only looks up in the specified list.

3 Variables should only return vectors

`qq()` only allows variables to return vectors. The whole string will be interpolated repeatedly according to longest vectors, and finally concatenated into a single long string.

```
x = 1:6
qqcat("@{x} is an @{ifelse(x %% 2, 'odd', 'even')} number.\n")

## 1 is an odd number.
## 2 is an even number.
## 3 is an odd number.
## 4 is an even number.
## 5 is an odd number.
## 6 is an even number.

y = c("a", "b")
z = c("A", "B", "C", "D", "E")
qqcat("@{x}, @{y}, @{z}\n")

## 1, a, A
## 2, b, B
## 3, a, C
## 4, b, D
## 5, a, E
## 6, b, A
```

This feature is especially useful if you want to generate a report such as formatted in a HTML table:

```
name = letters[1:4]
value = 1:4
qqcat("<tr><td>@{name}</td><td>@{value}</td><tr>\n")
```

```
## <tr><td>a</td><td>1</td><tr>
## <tr><td>b</td><td>2</td><tr>
## <tr><td>c</td><td>3</td><tr>
## <tr><td>d</td><td>4</td><tr>
```

The returned value can also be a vector while not collapsed into one string:

```
str = qq("@{x}, @{y}, @{z}", collapse = FALSE)
length(str)

## [1] 6

str

## [1] "1, a, A" "2, b, B" "3, a, C" "4, b, D" "5, a, E" "6, b, A"
```

4 You can also interpolate more complicated codes

Besides simple variables, you can also put chunk of codes in the string. Since `qq()` takes value of the last variable in the chunk as final returning value, you need to explicitly specify the return value in the chunk.

Following example is another way to contruct a HTML table in which the first row is assigned with class name.

```
name = letters[1:4]
value = 1:4
str = qq("`text = character(length(name))
  for(i in seq_along(name)) {
    if(i == 1) {
      text[i] = qq(\"<tr class='highlight'><td>@{name[i]}</td><td>@{value[i]}</td></tr>\n\")
    } else {
      text[i] = qq(\"<tr><td>@{name[i]}</td><td>@{value[i]}</td></tr>\n\")
    }
  }
text
`", code.pattern = "`CODE`")
```

```
cat(str)

## <tr class='highlight'><td>a</td><td>1</td></tr>
## <tr><td>b</td><td>2</td></tr>
## <tr><td>c</td><td>3</td></tr>
## <tr><td>d</td><td>4</td></tr>
```

In above example, the string also contains a `qq()` function, so please use different code patterns for them.

5 Simple template system

More advanced, you can make a simple template system. E.g., put following code in `template.html`. In the template, *R* codes are marked by `[%` and `%]`. Again, if `qq` is called inside the *R* codes in the template, you should use different code pattern instead of pattern in the template. That's why we use a complicated code pattern (`\\[%CODE%]\\`) in the template.

```
<html>
<body>
<h2>Report for [% report_name %]</h2>
<table>
<tr><th>name</th><th>value</th></tr>
[%
i = seq_along(name)
qq("<tr@{ifelse(i == 1, ' class=\"highlight\"', '')}><td>@{name}</td><td>@{value}</td></tr>
")
%]
</table>
</body>
</html>
```

Read `template.html` and do the interpolation:

```
template = paste(readLines("template.html"), collapse = "\n")

report_name = "test"
name = letters[1:4]
value = 1:4
html = qq(template, code.pattern = "\\[%CODE%\\]")
writeLines(html, con = "report.html")
```

You will get a report like:

```
<html>
<body>
<h2>Report for test</h2>
<table>
<tr><th>name</th><th>value</th></tr>
<tr class="highlight"><td>a</td><td>1</td></tr>
<tr><td>b</td><td>2</td></tr>
<tr><td>c</td><td>3</td></tr>
<tr><td>d</td><td>4</td></tr>

</table>
</body>
</html>
```