Introduction

Did you know that the `magrittr` pipe, `%>%`, can be used for more than just `data.frame`s and `tibble`s? In this blog post, we look at how we can create get and set functions for list elements.

Getting List Elements

First, let's create a simple list.

```
z1 <- list(a = pi, b = 2.718, c = 0.57721)
z1
# $a
# [1] 3.141593
#
# $b
# [1] 2.718
#
# $c
# [1] 0.57721
```

Let's say we want to access an element of this list, typically we would use the `[[` function to do so.

```
z1[[2]]
# [1] 2.718
```

But let's say we need to access this list as part of a chain using `magrittr`'s pipe operator, `%>%`. How can we do that? Well we can pipe our list into a `.` which acts as a placeholder for the list, on which we can perform our subset.

```
library(magrittr)
z1 %>% .[[2]]
# [1] 2.718
```

Another solution is to call `[[` using its syntactic form `[[()` using backticks (or quotes, see `?Quotes`).

```
z1 %>% `[[`(2)
# [1] 2.718
```

Admittedly, these two solutions don't look very nice. So what we can do instead is assign the `[[` function to an object which will, in effect, be a callable wrapper function.

```
get <- .Primitive("[[") # Equivalent to get <- `[[`
get(z1, 2)
# [1] 2.718
```

Primitives are functions that are internally implemented by R and so `.Primitive("[[")` tells R to dispatch to the underlying C code, which will be able to correctly identify which `[[` method to use on the list class (see `?.Primitive` for more details).

Since our list is now the first argument of `get()`, we have a much "cleaner" looking way of accessing elements of a list with the `magrittr` pipe operator than `[[`. And so, let's access the second element of our list using `get()` and the `magrittr` pipe.

```
z1 %>% get(2)
# [1] 2.718
```

We can also access the list using its names, too.

```
z1 %>% get("b")
# [1] 2.718
```

It even works with recursive indexing!

```
z2 <- list(a = list(b = 9, c = "hello"), d = 1:5)
z2
# $a
# $a$b
# [1] 9
#
# $a$c
# [1] "hello"
#
#
# $d
# [1] 1 2 3 4 5
z2 %>% get(c("a", "c")) # equivalent to z %>% get(c(1, 2))
# [1] "hello"
```

Note, you may want to choose a better name than `get` to avoid clashes with the `base::get()` function.

Setting List Elements

Similarly we can create a `set()` function to assign values to elements of our list using `.Primitive("[[<-")`. Let’s add a fourth element to our list.

```
set <- .Primitive("[[<-")
z1 <- z1 %>% set("d", 4.6692)
z1
# $a
# [1] 3.141593
#
# $b
# [1] 2.718
#
# $c
# [1] 0.57721
#
# $d
# [1] 4.6692
```

And now just as `set()` giveth, `set()` taketh away.

```
z1 <- z1 %>% set("d", NULL)
z1
# $a
# [1] 3.141593
#
# $b
# [1] 2.718
#
# $c
# [1] 0.57721
```

Of course as this is a list, we can set any kind of data.

```
z1 %>% set("data", data.frame(a = c(1, 2, 2, 4), b = c(2, 3, 7, 4)))
# $a
# [1] 3.141593
#
# $b
# [1] 2.718
#
# $c
# [1] 0.57721
#
# $data
#   a b
# 1 1 2
# 2 2 3
# 3 2 7
# 4 4 4
```

Or even overwrite elements.

```
z1 %>% set("b", 4.6692)
# $a
# [1] 3.141593
#
# $b
# [1] 4.6692
#
# $c
# [1] 0.57721
```

Conclusion

This was just a short blog post to highlight the power of `magrittr` in combination with R primitives. We also saw how to rewrite and manipulate syntactic forms of internal R functions. What other interesting use cases have you found for the `magrittr` pipe?