Advanced R by Hadley Wickham

Chapter 18: Expressions

Jake Riley

Dec. 3, 2020

Chapter 18

metaprogramming: treating code as data

Libraries:

```
library(tidyverse)
library(rlang)
```

Agenda:

- Expressions
- Abstract Syntax Trees (AST)
- Symbols
- Calls
- Parsing
- Live demos

Expressions

If you try to run this in a new session, you will get an error as x isn't defined

```
x * 10
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

As an expression, we can interact with code without running it

```
z <- expr(x * 10)
z
```

```
## x * 10
```

The code isn't evaluated until we use `eval()`

```
x <- 4
eval(z)
```

```
## [1] 40
```

Evaluating multiple expressions

We can write multiple expressions at once & it acts similar to `source()`

`expression()` returns a vector and can be passed to `eval()`

```
eval(expression(
  x <- 4,
  x * 10
))
```

```
## [1] 40
```

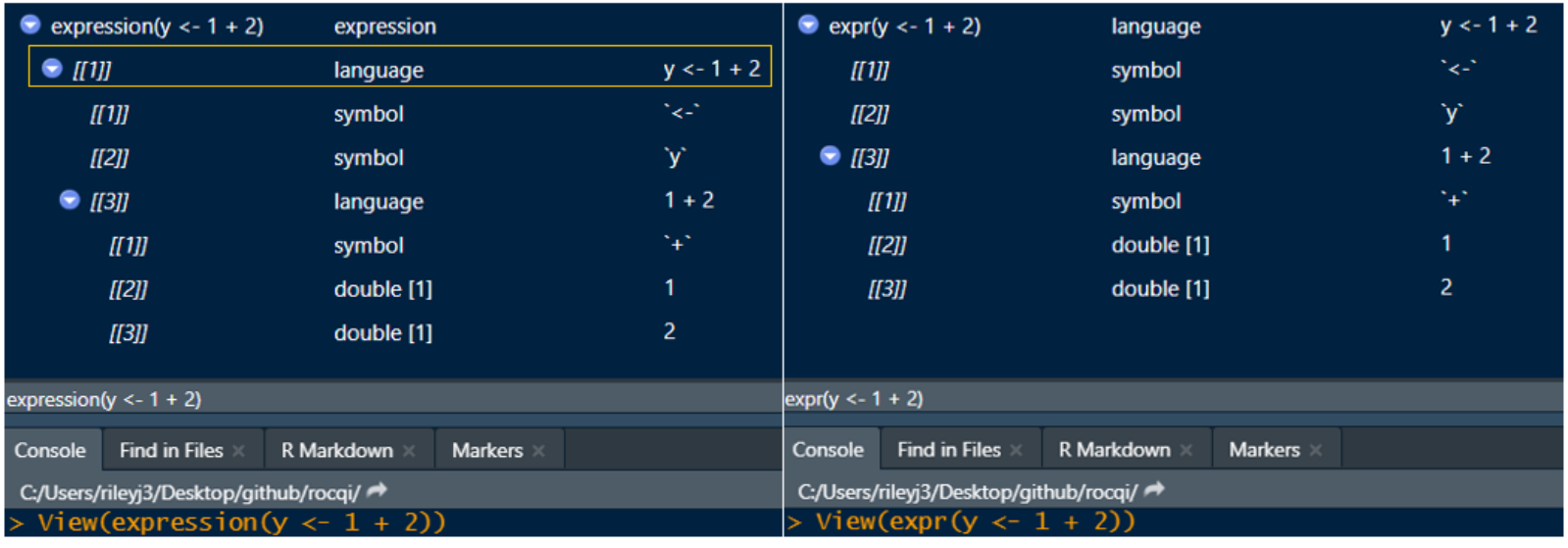
`exprs()` does not and has to be used in a loop

```
for (i in  exprs(x <- 4,x * 10)) {
  print(i)
  print(eval(i))
}
```

```
## x <- 4
## [1] 4
## x * 10
## [1] 40
```

`expression()` vs `expr()`

- `expression()` is ready for evaluation
- `expr()` is ready for manipulation



18.2.3 Ways to write

- **infix call:** function is between objects, is evaluated

```
x <- 5 + 10
```

- **prefix call:** function is in front of objects, is evaluated most common with functions

```
mean(5:10)
`+`(5, 10)
`<-`(x, `+`(5, 10))
```

- **expression:** the code as written, not evaluated class = "expression"

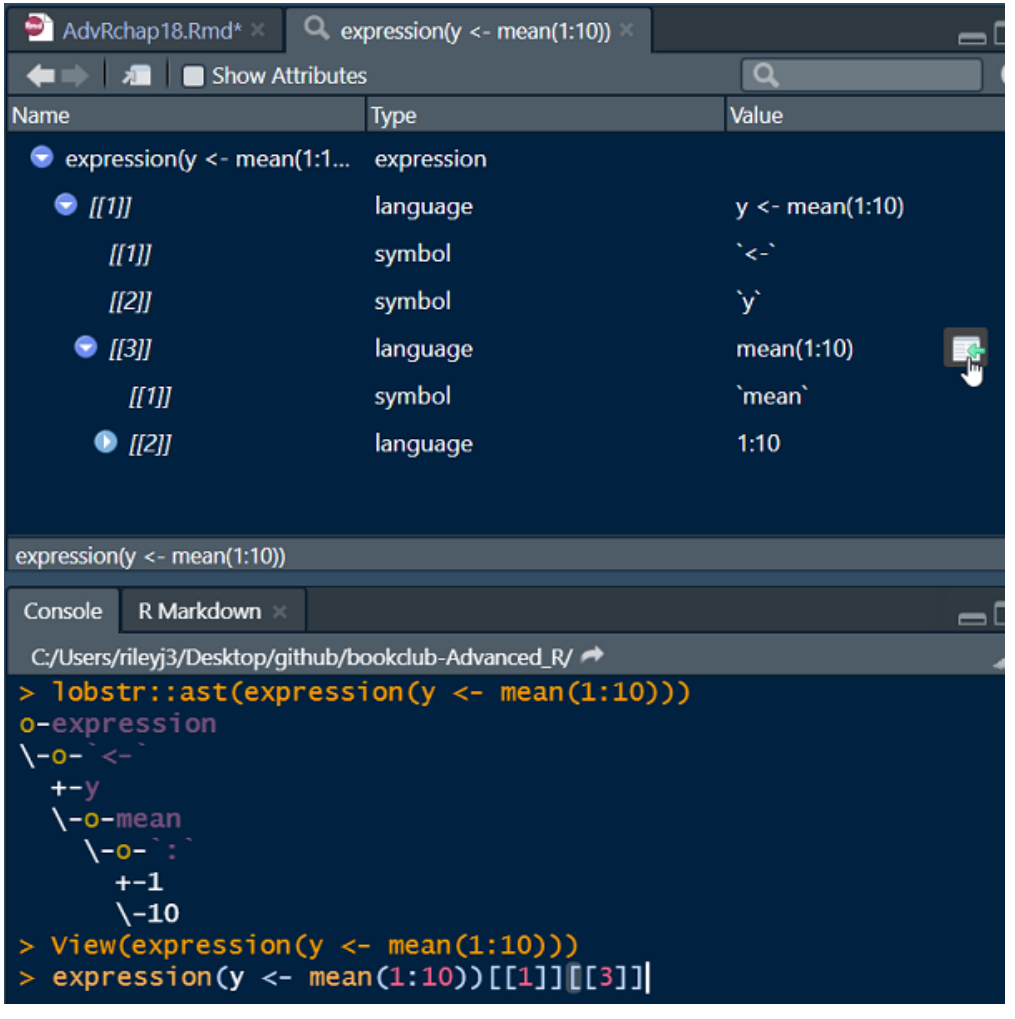
```
expression(x <- 5 + 10)
```

- **call:** uses a named function followed by it's arguments, is not evaluated class = "language"

```
call("+", 5, 10)
```

Abstract Syntax Trees

- `lobstr::ast()` is nice but I prefer `View()`



Replacing code

```
z <- expression(mean(1:10))
# View(z)
eval(z)
```

```
## [1] 5.5
```

```
z[[1]][[1]]
```

```
## mean
```

```
z[[1]][[1]] <- sum

# View(z)
eval(z)
```

```
## [1] 55
```

18.3.2 Symbols

symbol is the "name" of an object, ex: `mtcars`, `mean`, etc

You can convert a **string to a symbol** with `rlang::sym()`

I use this a lot when debugging tidy evaluation

```
  x <- rlang::sym("hwy")

head(mpg) %>%
  select_if(is.numeric) %>%
  mutate('{{x}}_kpl' := {{x}} * 0.425)
```

displ	year	cyl	cty	hwy	hwy_kpl
1.8	1999	4	18	29	12.325
1.8	1999	4	21	29	12.325
2.0	2008	4	20	31	13.175
2.0	2008	4	21	30	12.750
2.8	1999	6	16	26	11.050
2.8	1999	6	18	26	11.050

You can convert **a symbol to a string** with

`as.character(expression(hwy))`

or `rlang::as_string(rlang::expr(hwy))`

18.3.3 Standardized Calls

- `call_standardise()` is almost the same as `match_call()`

```
new_msg <- function(data, digits = 5, ...) {
  message(
    paste(
      match.call()$data,
      "has",
      nrow(data),
      "rows"
    )
  )
}
```

```
new_msg(digits = 3, mpg)
```

```
## mpg has 234 rows
```

```
new_msg(d = mpg)
```

```
## Error in new_msg(d = mpg): argument 1 matches multiple formal arguments
```

```
new_msg(da = mpg)
```

```
## mpg has 234 rows
```

Can get a heads up about this with

```
options(warnPartialMatchArgs = TRUE)
```

18.4 Parsing

`parse_exprs()` is the same as `parse(text = ...)`

```
x3 <- "a <- 1; a + 1"

parse(text = x3)
```

```
## expression(a <- 1, a + 1)
```

```
as.list(parse(text = x3))
```

```
## [[1]]
## a <- 1
##
## [[2]]
## a + 1
```

```
rlang::parse_exprs(x3)
```

```
## [[1]]
## a <- 1
##
## [[2]]
## a + 1
```

You can deparse with `expr_text()`

```
z <- expr(y <- x + 10)
expr_text(z)
```

```
## [1] "y <- x + 10"
```

Conclusion

- `rlang` functions are more "ready to go"
- have protective features to bring back vectors of the same length as the input (`parse_exprs()` vs `deparse()`)
- shinyobjects