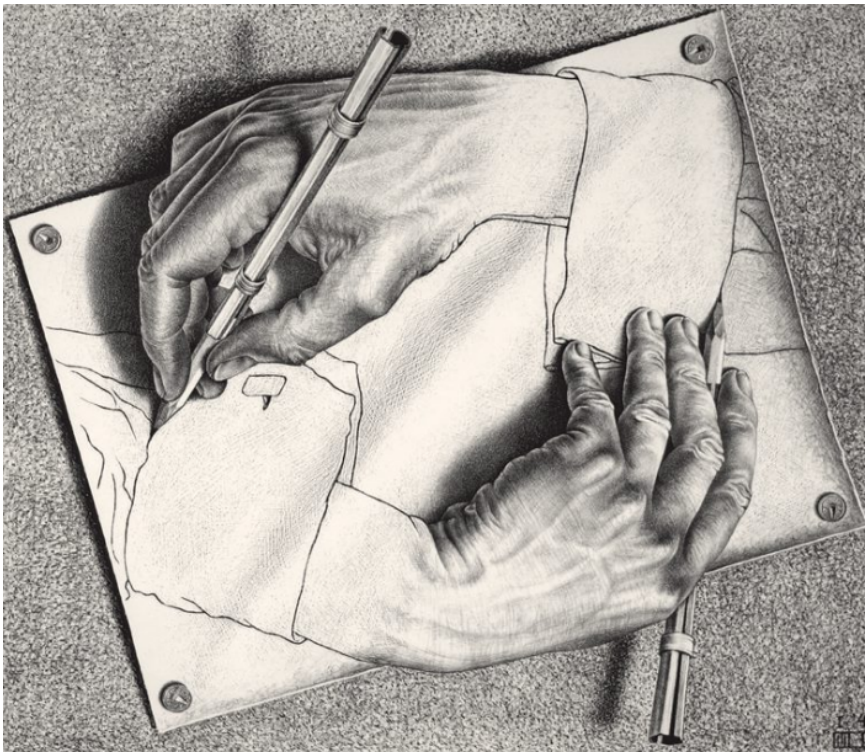
Contents


```
library(rlang)
library(tidyverse)
library(magrittr)
library(pipeR)

## this function gets the arguments passed to a function
get_args <- function() {
    as.list(match.call(
        def = sys.function(-1),
        call = sys.call(-1)))[-1]
}
```

1 A Spoonful of Metaprogramming

(is good for what ails you)



2 What?

A language that supports metaprogramming is a language that supports programming itself. This could mean many things, but here we are going to focus on 2 aspects:

2.1 Code is data, data is code

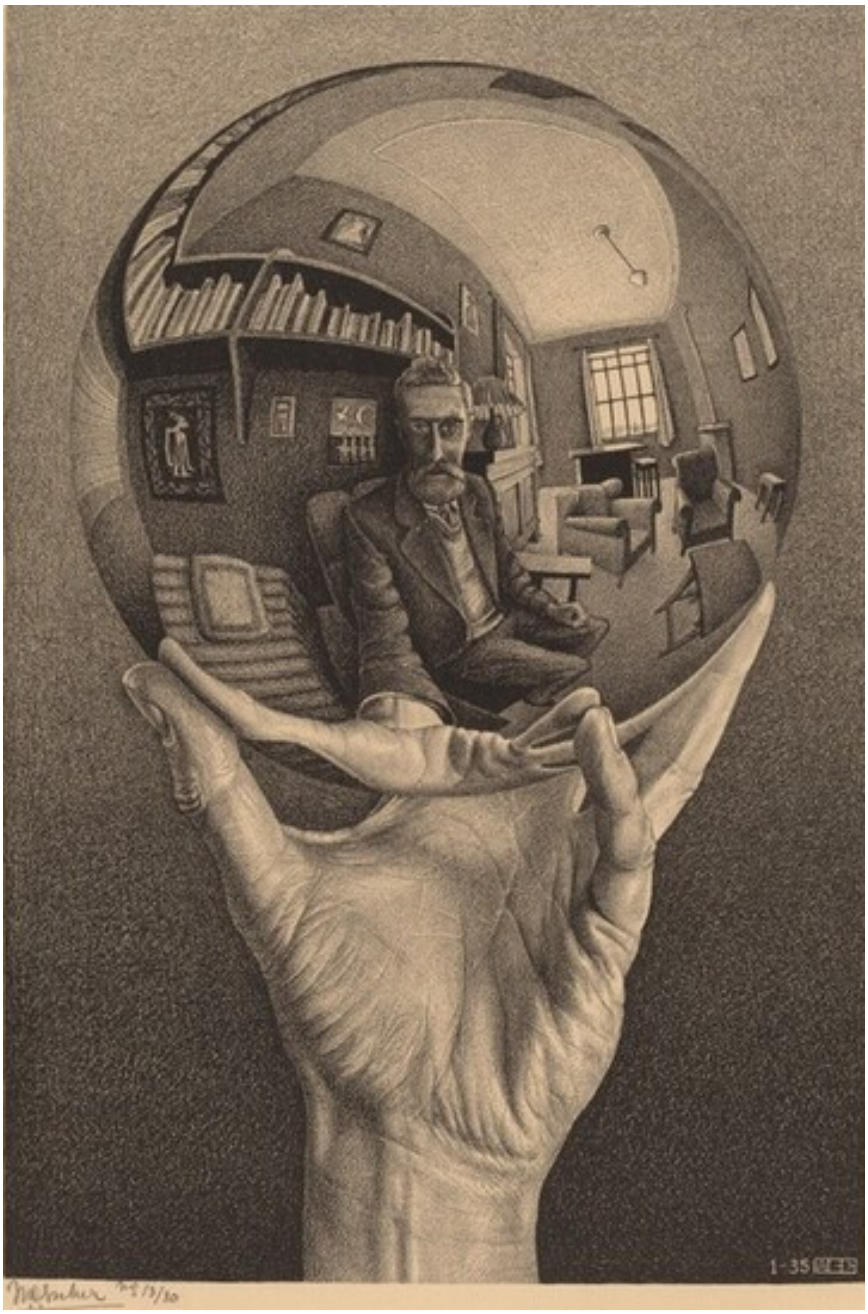
2.2 The programming language itself is programmable

2.3 Why?

fun mostly

...and to expand the number of operations we can perform without thinking about them

3 Manipulating Functions



There are 3 parts of a function in R:

```
f <- function(x) x + 25

## formals
formals(f)

## body
body(f)

## environment
environment(f)
```

3.1 Formals

Formals are a list, so we can treat it as such:

```
## define multi-parameter function
(g <- function(x, y, z) (x + y) / z)

g(1, 2, 3)

## look at formals
formals(g)

## mess with it
rev_forms <- function(fun) {
    formals(fun) <- rev(formals(fun))
    fun
}

(g <- rev_forms(g))

g

g(1, 2, 3)

## make it more interesting : add some variables
formals(g) %<>% append(list("a" = missing_arg(),
                            "b" = missing_arg(),
                            "c" = missing_arg()))
```

```
g

## add global scope variables
w <- runif(1)

formals(g)$c <- w

g
```

3.2 Body

The function body is an expression

```
## look at body
body(g)

## see how the data structure is put together
body(g) %>% as.character

body(g)[[1]]
body(g)[[2]]
body(g)[[3]]

## modify it
body(g)[[3]] <- quote(z + a + b + c)

body(g)

g

## evaluate our new expression
g(1, 2, 3)

## let c be evaluated by w
g(z = 1,
  y = 2,
  x = 3,
  a = 4,
  b = 5)
```

```
## let c = 6
g(1, 2, 3, 4, 5, 6)

## check that it matches!
(3 + 2) / (1 + 4 + 5 + 6)
```

We can even have a function find its own definition:

```
f <- function(x = 5) {
    y <- x + 10
    ## return definition of itself
    sys.function()
}

f()

body(f)

body(f)[[1]]
body(f)[[2]]
body(f)[[3]]

## what if we set our body of f to our body of f?
body(f)[[2]] <- body(f)

body(f)

f()

f <- function(x = 5) {
    if (x == 0) sys.function()
    else {
        body(f)[[3]] <- body(f)
        f(x - 1)
    }
}

f(10)
```

3.3 Environments

Environments are the scope in which the function is evaluated

```
## our current environment
environment()

## scope inside of the function
h <- function() {
    environment()
}

h()

## get parent environment
parent.env(h())

## these can be manipulated like lists, but they are really more like a hash table wit
new_env <- new.env()
new_env$x <- 1
new_env$x

## we can loop through and assign
map2(state.abb, 1:50, ~ {new_env[[.x]] <- .y})

## access each value by name
new_env$MA
new_env$RI

## and get values as needed
new_env %>% names()
```

4 Manipulating Expressions

We can capture code without evaluating it using expr()

```
x <- 2
y <- 0

## quote!
(z <- expr(y <- x * x * x * x * x))

z %>% typeof

## unquote
eval(z)

y

## we can also selectively quote subexpressions
(z <- bquote(y <- .(x) + 256))

y

eval(z)

y
```

What was done above is called quotation (the act of capturing an unevaluated expression) and unquotation (the ability to evaluate parts of an otherwise quoted expression). Together, this is referred to as **Quasiquotation**.

This is all over the place in R

```
## ever wonder why we can write this
mtcars %>%
    filter(qsec > 20)

## and not have to do this?
mtcars %>% filter(.[["qsec"]] > 20)

## or this?
mtcars %>% filter(mtcars$qsec > 20)

## especially since
qsec
```

Under the hood, we are taking the expression, and waiting to evaluate it until we have the proper environment to evaluate it in. In this case, our awaited environment is the dataset mtcars.

This is an example of tidy evaluation. It combines quasiquotation, quosures (a data structure that captures an expression and its environment), and data masks (which allow an expression to be evaluated in the context of its dataset).

5 TidyEval

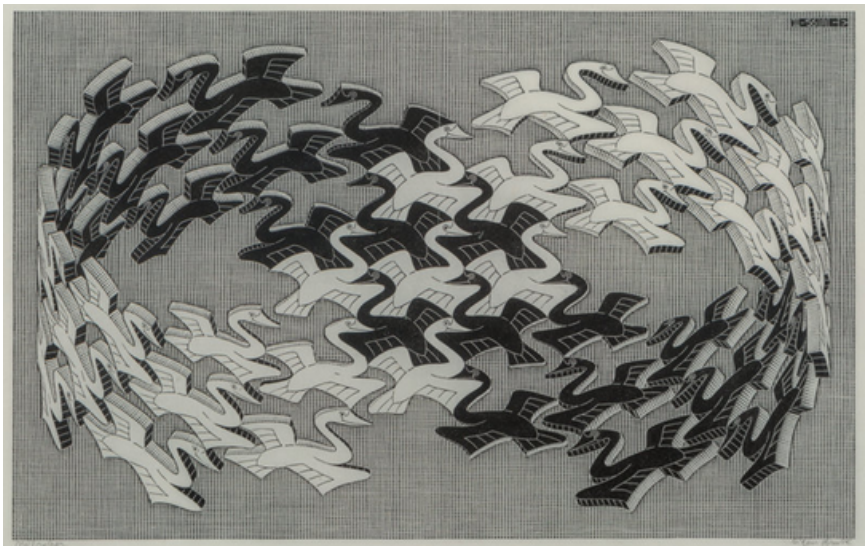
Tidyeval is very practical, as it lets you use functions without quotes all over the place as well as reference things in different scopes and data masks. It helps match expressions to environments. It helps create a more interactive workflow at the expense of having to know a bunch more things when writing functions.

Pragmatically, you can use it to write functions that act on nested data (i.e., data in dataframes)

```
## if you're only doing 1 thing, you can use {{}}
tally_it <- function(.data, column) {
    .data %>%
        group_by({{column}}) %>%
        tally(sort = TRUE)
}

mtcars %>% tally_it(mpg)
mtcars %>% tally_it(mpg, hp)

## you can use ... parsing and expansion for multiple args
tally_it <- function(.data, ...) {
    (args <- enexprs(...))

    .data %>%
        group_by(!!!args) %>%
        tally(sort = TRUE)
}

mtcars %>% tally_it(mpg, hp, disp, cyl)

## you can even coerce strings to exprs beforehand
tally_it <- function(.data, ...) {
    args <- enexprs(...)

    if (is.character(pluck(args, 1)))
        args <- map(args, parse_expr)

    .data %>%
        group_by(!!!args) %>%
        tally(sort = TRUE)
}

mtcars %>% tally_it("mpg", "hp", "disp")

## plots
plot_it <- function(.data, x, y) {
    ## get arguments and coerce to character
    xlab <- as.character(get_args()$x)
    ylab <- as.character(get_args()$y)
```

```
    .data %>%
        ggplot(aes(x = {{x}}, y = {{y}})) +
        geom_point() +
        ggtitle(paste0(ylab, " ~ ", xlab))
}

mtcars %>% plot_it(mpg, disp)
mtcars %>% plot_it(mpg, hp)
```

6 Substitutions



This is useful because we can do similar substitutions, but with all code

```
## we can coerce exprs to strings and manipulate them
expr(y <- x + x + x + x + z) %>%
    deparse() %>%
    str_glue(" + z + z + z") %>%
    parse_expr() -> new_expression

x <- 2
y <- 0
z <- 1
```

```
eval(new_expression)

y
```

This idea means we can now make code 'data'

```
## preprocessing
(mtcars %<>%
    ## get car names
    as_tibble(rownames = "car_names"))

## generate a bunch of statements
mtcars %>%
    pull(1) %>%
    ## get just makes
    str_extract("^[A-Za-z]+") %>%
    unique() %>>%
    ## set names
    (~ unique_names) %>%
    ## 'build' out a string
    map(~ .x %>% paste0("ifelse(str_detect(car_names, \"", ., "\"), TRUE, FALSE)") %>%
    set_names(unique_names) -> conditional_statements

mtcars %>%
    mutate(!!!conditional_statements) %>%
    glimpse
```

We can **quote** data to turn it into an expression, and then **unquote** it to make it into code

We can do more with this by writing code that changes R code. In other languages, these are often called **macros**.

```
## in R we often chain together pipelines like so
mtcars %>%
    select(mpg, cyl, disp, hp) %>%
    filter(cyl == 6) %>%
    summarize(mean(mpg))

## this translates to the following
summarize(
```

```
    filter(
        select(mtcars, mpg, cyl, disp, hp), cyl == 6), mean(mpg))
```

The pipe is an example of an infix operator. It takes arguments on both sides, and translates the code to a properly written function call. We can write our own:

```
`%r%` <- function(expr, num) replicate(num, expr)

## now we can use this like we would any other infix operator
rexp(1) %r% 3
rexp(3) %r% 3
rexp(3) %r% 3 %r% 4
```

6.1 An example : Replicating Clojure Threading Macros

While the pipe is fantastic, it does replicate %>% over and over again at the end of each line. This is not that bad if you have a key-binding

In the clojure programming language, they use prefix notation:

```
;; prefix notation
(+ 1 2 3 4 5)

;; this is the equivalent pipe
(-> (load-data "xyz.csv")
    ($ "those_columns")
    (filter {:cyl {:eq 6}})
    (mean :mpg))
```

We can create something similar in R minus all those parentheses:

```
p_ <- function(x, ...) {
    ## take in all our arguments as quoted expressions
    enexprs(x, ...) %>%
        ## deparse them to turn them into strings! paste the pipe
        map_chr(~ deparse(.x) %>% str_glue(" %>% ")) %>%
        ## collapse to one string
        paste(collapse = "") %>%
        ## add identity to the end (lazy hack)
        str_glue("identity") %>%
        ## parse the string back to a quoted expression
```

```
        parse(text = .) %>%
        ## evaluate it!
        eval()
}

plus_n <- function(n) function(x) x + n

plus_n(3)(2)

x <- 1

p_(x,
   plus_n(3)(),
   plus_n(2)(),
   plus_n(3)())

p_(mtcars,
   select(mpg, cyl, disp, hp),
   filter(cyl == 6),
   summarize(mean(mpg)))

## walk through
## take in all our arguments as quoted expressions
exprs(mtcars,
        select(mpg, cyl, disp, hp),
        filter(cyl == 6),
        summarize(mean(mpg))) %>%
    ## deparse them to turn them into strings! paste the pipe
    map_chr(~ deparse(.x) %>% str_glue(" %>% ")) %>%
    ## collapse to one string
    paste(collapse = "") %>%
    ## add identity to the end (lazy hack)
    str_glue("identity") %>%
    ## parse the string back to a quoted expression
    parse(text = .) %>%
    ## evaluate it!
    eval()
```

Of course here all we did was replace %>% with ,

The big idea is that we are not delegated to writing code ac-

cording to the whims of the language designer, but we are the language designer

7 Generating Code with R

It doesn't even need to be R code that we generate. We can use R's flexibility to create other types of code as well. A great example is **dbplyr**, which translates R dplyr code to SQL.

```
library(dbplyr)
```

```
## generate sql
mtcars %>%
    lazy_frame(con = simulate_mssql()) %>%
    select(mpg, cyl, disp, hp) %>%
    filter(cyl == 6) %>%
    summarize(mean(mpg)) %>%
    show_query()

## combine with metaprogramming, using sql primitives
## generate a bunch of statements
mtcars %>%
    pull(1) %>%
    ## get just makes
    str_extract("^[A-Za-z]+") %>%
    unique() %>>%
    ## set names
    (~ unique_names) %>%
    ## 'build' out a string
    map(~ .x %>% paste0("ifelse(car_names %like% \"", ., "\", TRUE, FALSE)") %>% parse
    set_names(unique_names) -> conditional_statements

## generate sql
mtcars %>%
    lazy_frame(con = simulate_mssql()) %>%
    mutate(!!!conditional_statements) %>%
    show_query()
```

Another great example is shiny, which generates html / css / javascript.

```
library(shiny)

(ui <- fluidPage(
    titlePanel("title panel"),
    sidebarLayout(
        sidebarPanel("sidebar panel"),
        mainPanel("main panel"))))

div(h1(title()))
sidebarPanel("Panel")
mainPanel("Main Panel")
```

8 Other things to check out

tidyeval rlang how calls work how formulas work how symbols work how interpreters work how compilers work

8.1 Resources to look into

Advanced R Thomas Mailund's books on R Lisp family programming languages The Documentation (tm)