Function operators

In this chapter, you'll learn about function operators (FOs). A function operator is a function that takes one (or more) functions as input and returns a function as output. In some ways, function operators are similar to functionals: there's nothing you can't do without them, but they can make your code more readable and expressive, and they can help you write code faster. The main difference is that functionals extract common patterns of loop use, where function operators extract common patterns of anonymous function use.

The following code shows a simple function operator, `chatty()`. It wraps a function, making a new function that prints out its first argument. It's useful because it gives you a window to see how functionals, like `vapply()`, work.

```
chatty <- function(f) {
  function(x, ...) {
    res <- f(x, ...)
    cat("Processing ", x, "\n", sep = "")
    res
  }
}
f <- function(x) x ^ 2
s <- c(3, 2, 1)
chatty(f)(1)
#> Processing 1
#> [1] 1

vapply(s, chatty(f), numeric(1))
#> Processing 3
#> Processing 2
#> Processing 1
#> [1] 9 4 1
```

In the last chapter, we saw that many built-in functionals, like `Reduce()`, `Filter()`, and `Map()`, have very few arguments, so we had to use anonymous functions to modify how they worked. In this chapter, we'll build specialised substitutes for common anonymous functions that allow us to communicate our intent more clearly. For example, in multiple inputs we used an anonymous function with `Map()` to supply fixed arguments:

```
Map(function(x, y) f(x, y, zs), xs, ys)
```

Later in this chapter, we'll learn about partial application using the `partial()`function. Partial application encapsulates the use of an anonymous function to supply default arguments, and allows us to write succinct code:

```
Map(partial(f, zs = zs), xs, yz)
```

This is an important use of FOs: by transforming the input function, you eliminate parameters from a functional. In fact, as long as the inputs and outputs of the function remain the same, this approach allows your functionals to be more

extensible, often in ways you haven't thought of.

The chapter covers four important types of FO: behaviour, input, output, and combining. For each type, I'll show you some useful FOs, and how you can use as another to decompose problems: as combinations of multiple functions instead of combinations of arguments. The goal is not to exhaustively list every possible FO, but to show a selection that demonstrate how they work together with other FP techniques. For your own work, you'll need to think about and experiment with how function operators can help you solve recurring problems.

Outline

- Behavioural FOs introduces you to FOs that change the behaviour of a function like automatically logging usage to disk or ensuring that a function is run only once.
- Output FOs shows you how to write FOs that manipulate the output of a function. These can do simple things like capturing errors, or fundamentally change what the function does.
- Input FOs describes how to modify the inputs to a function using a FO like `Vectorize()` or `partial()`.
- Combining FOs shows the power of FOs that combine multiple functions with function composition and logical operations.

Prerequisites

As well as writing FOs from scratch, this chapter uses function operators from the memoise, plyr, and pryr packages. Install them by running `install.packages(c("memoise", "plyr", "pryr"))`.

Behavioural FOs

Behavioural FOs leave the inputs and outputs of a function unchanged, but add some extra behaviour. In this section, we'll look at functions which implement three useful behaviours:

- Add a delay to avoid swamping a server with requests.
- Print to console every n invocations to check on a long running process.
- Cache previous computations to improve performance.

To motivate these behaviours, imagine we want to download a long vector of URLs. That's pretty simple with `lapply()` and `download_file()`:

```
download_file <- function(url, ...) {
  download.file(url, basename(url), ...)
}
lapply(urls, download_file)
```

(`download_file()` is a simple wrapper around `utils::download.file()` which provides a reasonable default for the file name.)

There are a number of useful behaviours we might want to add to this function. If the list was long, we might want to print a `.` every ten URLs so we know that the function's still working. If we're downloading files over the internet, we might want to add a small delay between each request to avoid hammering the server. Implementing these behaviours in a for loop is rather complicated. We can no longer use `lapply()` because we need an external counter:

```
i <- 1
for(url in urls) {
  i <- i + 1
  if (i %% 10 == 0) cat(".")
  Sys.delay(1)
  download_file(url)
}
```

Understanding this code is hard because different concerns (iteration, printing, and downloading) are interleaved. In the remainder of this section we'll create FOs that encapsulate each behaviour and allow us to write code like this:

```
lapply(urls, dot_every(10, delay_by(1, download_file)))
```

Implementing `delay_by()` is straightforward, and follows the same basic template that we'll see for the majority of FOs in this chapter:

```
delay_by <- function(delay, f) {
  function(...) {
    Sys.sleep(delay)
    f(...)
  }
}
system.time(runif(100))
#>    user  system elapsed
#>   0.001   0.000   0.000
system.time(delay_by(0.1, runif)(100))
#>    user  system elapsed
#>   0.000   0.000   0.101
```

`dot_every()` is a little bit more complicated because it needs to manage a counter. Fortunately, we saw how to do that in mutable state.

```
dot_every <- function(n, f) {
  i <- 1
  function(...) {
    if (i %% n == 0) cat(".")
    i <<- i + 1
    f(...)
  }
}
x <- lapply(1:100, runif)
x <- lapply(1:100, dot_every(10, runif))
#> ..........
```

Notice that I've made the function the last argument in each FO. This makes it easier to read when we compose multiple function operators. If the function were the first argument, then instead of:

```
download <- dot_every(10, delay_by(1, download_file))
```

we'd have

```
download <- dot_every(delay_by(download_file, 1), 10)
```

That's harder to follow because (e.g.) the argument of `dot_every()` is far away from its call. This is sometimes called the Dagwood sandwich problem: you have too much filling (too many long arguments) between your slices of bread (parentheses).

I've also tried to give the FOs descriptive names: delay by 1 (second), (print a) dot every 10 (invocations). The more clearly the function names used in your code express your intent, the easier it will be for others (including future you) to read and understand the code.

Memoisation

Another thing you might worry about when downloading multiple files is accidentally downloading the same file multiple times. You could avoid this by calling `unique()` on the list of input URLs, or manually managing a data structure that mapped the URL to the result. An alternative approach is to use memoisation: modify a function to automatically cache its results.

```
library(memoise)
```

```
slow_function <- function(x) {
  Sys.sleep(1)
  10
}
system.time(slow_function())
#>    user  system elapsed
#>   0.000   0.000   1.001
system.time(slow_function())
#>    user  system elapsed
#>   0.000   0.000   1.001
fast_function <- memoise(slow_function)
system.time(fast_function())
#>    user  system elapsed
#>   0.000   0.000   1.002
system.time(fast_function())
#>    user  system elapsed
#>   0.000   0.000   0.001
```

Memoisation is an example of the classic computer science tradeoff of memory versus speed. A memoised function can

run much faster because it stores all of the previous inputs and outputs, using more memory.

A realistic use of memoisation is computing the Fibonacci series. The Fibonacci series is defined recursively: the first two values are 1 and 1, then f(n) = f(n - 1) + f(n - 2). A naive version implemented in R would be very slow because, for example, `fib(10)` computes `fib(9)` and `fib(8)`, and `fib(9)` computes `fib(8)` and `fib(7)`, and so on. As a result, the value for each value in the series gets computed many, many times. Memoising `fib()` makes the implementation much faster because each value is computed only once.

```
fib <- function(n) {
  if (n < 2) return(1)
  fib(n - 2) + fib(n - 1)
}
system.time(fib(23))
#>    user  system elapsed
#>   0.125   0.006   0.132
system.time(fib(24))
#>    user  system elapsed
#>   0.194   0.003   0.196

fib2 <- memoise(function(n) {
  if (n < 2) return(1)
  fib2(n - 2) + fib2(n - 1)
})
system.time(fib2(23))
#>    user  system elapsed
#>   0.004   0.000   0.004
system.time(fib2(24))
#>    user  system elapsed
#>   0.001   0.000   0.000
```

It doesn't make sense to memoise all functions. For example, a memoised random number generator is no longer random:

```
runifm <- memoise(runif)
runifm(5)
#> [1] 0.5374603 0.3466054 0.3079437 0.3631236 0.9511340
runifm(5)
#> [1] 0.5374603 0.3466054 0.3079437 0.3631236 0.9511340
```

Once we understand `memoise()`, it's straightforward to apply to our problem:

```
download <- dot_every(10, memoise(delay_by(1, download_file)))
```

This gives a function that we can easily use with `lapply()`. However, if something goes wrong with the loop inside `lapply()`, it can be difficult to tell what's going on. The next section will show how we can use FOs to pull back the

curtain and look inside.

Capturing function invocations

One challenge with functionals is that it can be hard to see what's going on inside of them. It's not easy to pry open their internals like it is with a for loop. Fortunately we can use FOs to peer behind the curtain with `tee()`.

`tee()`, defined below, has three arguments, all functions: `f`, the function to modify;`on_input`, a function that's called with the inputs to `f`; and `on_output`, a function that's called with the output from `f`.

```
ignore <- function(...) NULL
tee <- function(f, on_input = ignore, on_output = ignore) {
  function(...) {
    on_input(...)
    output <- f(...)
    on_output(output)
    output
  }
}
```

(The function is inspired by the unix shell command `tee`, which is used to split up streams of file operations so that you can both display what's happening and save intermediate results to a file.)

We can use `tee()` to look inside the `uniroot()` functional, and see how it iterates its way to a solution. The following example finds where `x` and `cos(x)` intersect:

```
g <- function(x) cos(x) - x
zero <- uniroot(g, c(-5, 5))
show_x <- function(x, ...) cat(sprintf("%+.08f", x), "\n")

# The location where the function is evaluated:
zero <- uniroot(tee(g, on_input = show_x), c(-5, 5))
#> -5.00000000
#> +5.00000000
#> +0.28366219
#> +0.87520341
#> +0.72298040
#> +0.73863091
#> +0.73908529
#> +0.73902425
#> +0.73908529
# The value of the function:
zero <- uniroot(tee(g, on_output = show_x), c(-5, 5))
#> +5.28366219
#> -4.71633781
#> +0.67637474
```

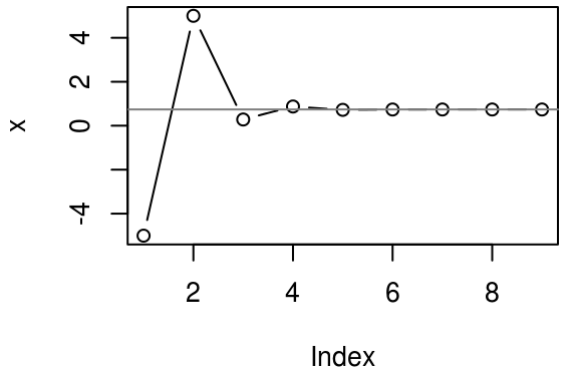
```
#> -0.23436269
#> +0.02685676
#> +0.00076012
#> -0.00000026
#> +0.00010189
#> -0.00000026
```

`cat()` allows us to see what's happening as the function runs, but it doesn't give us a way to work with the values after the function as completed. To do that, we could capture the sequence of calls by creating a function, `remember()`, that records every argument called and retrieves them when coerced into a list. The small amount of S3 code needed is explained in S3.

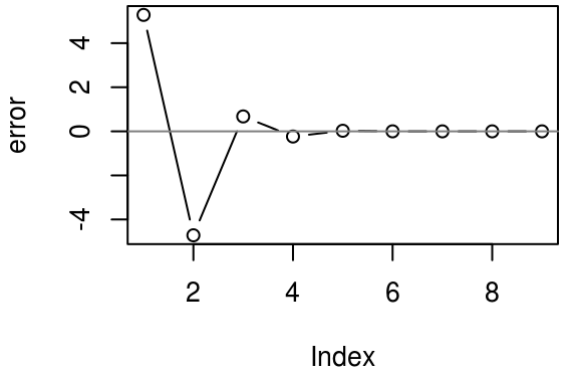
```
remember <- function() {
  memory <- list()
  f <- function(...) {
    # This is inefficient!
    memory <<- append(memory, list(...))
    invisible()
  }

  structure(f, class = "remember")
}
as.list.remember <- function(x, ...) {
  environment(x)$memory
}
print.remember <- function(x, ...) {
  cat("Remembering...\n")
  str(as.list(x))
}
```

Now we can draw a picture showing how uniroot zeroes in on the final answer:

```
locs <- remember()
vals <- remember()
zero <- uniroot(tee(g, locs, vals), c(-5, 5))
x <- unlist(as.list(locs))
error <- unlist(as.list(vals))
plot(x, type = "b"); abline(h = 0.739, col = "grey50")
```




```
plot(error, type = "b"); abline(h = 0, col = "grey50")
```



Laziness

The function operators we've seen so far follow a common pattern:

```
funop <- function(f, otherargs) {
  function(...) {
    # maybe do something
    res <- f(...)
    # maybe do something else
    res
  }
}
```

Unfortunately there's a problem with this implementation because function arguments are lazily evaluated: `f()` may have changed between applying the FO and evaluating the function. This is a particular problem if you're using a for loop or`lapply()` to apply multiple function operators. In the following example, we take a list of functions and delay each one. But when we try to evaluate the mean, we get the sum instead.

```
funs <- list(mean = mean, sum = sum)
funs_m <- lapply(funs, delay_by, delay = 0.1)

funs_m$mean(1:10)
```

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

We can avoid that problem by explicitly forcing the evaluation of `f()`:

```
delay_by <- function(delay, f) {
  force(f)
  function(...) {
    Sys.sleep(delay)
    f(...)
  }
}

funs_m <- lapply(funs, delay_by, delay = 0.1)
funs_m$mean(1:10)
#> [1] 5.5
```

It's good practice to do that whenever you create a new FO.