Closures

> “An object is data with functions. A closure is a function with data.” — John D. Cook

One use of anonymous functions is to create small functions that are not worth naming. Another important use is to create closures, functions written by functions. Closures get their name because they **enclose** the environment of the parent function and can access all its variables. This is useful because it allows us to have two levels of parameters: a parent level that controls operation and a child level that does the work.

The following example uses this idea to generate a family of power functions in which a parent function (`power()`) creates two child functions (`square()` and `cube()`).

```
power <- function(exponent) {
  function(x) {
    x ^ exponent
  }
}

square <- power(2)
square(2)
#> [1] 4
square(4)
#> [1] 16

cube <- power(3)
cube(2)
#> [1] 8
cube(4)
#> [1] 64
```

When you print a closure, you don’t see anything terribly useful:

```
square
#> function(x) {
#>     x ^ exponent
#>   }
#> <environment: 0x3a50680>
cube
#> function(x) {
#>     x ^ exponent
#>   }
#> <environment: 0x37f10f8>
```

That's because the function itself doesn't change. The difference is the enclosing environment, `environment(square)`. One way to see the contents of the environment is to convert it to a list:

```
as.list(environment(square))
#> $exponent
#> [1] 2
as.list(environment(cube))
#> $exponent
#> [1] 3
```

Another way to see what's going on is to use `pryr::unenclose()`. This function replaces variables defined in the enclosing environment with their values:

```
library(pryr)
unenclose(square)
#> function (x)
#> {
#>     x^2
#> }
unenclose(cube)
#> function (x)
#> {
#>     x^3
#> }
```

The parent environment of a closure is the execution environment of the function that created it, as shown by this code:

```
power <- function(exponent) {
  print(environment())
  function(x) x ^ exponent
}
zero <- power(0)
#> <environment: 0x4197fa8>
environment(zero)
#> <environment: 0x4197fa8>
```

The execution environment normally disappears after the function returns a value. However, functions capture their enclosing environments. This means when function a returns function b, function b captures and stores the execution environment of function a, and it doesn't disappear. (This has important consequences for memory use, see memory usage for details.)

In R, almost every function is a closure. All functions remember the environment in which they were created, typically either the global environment, if it's a function that you've written, or a package environment, if it's a function that someone else has written. The only exception is primitive functions, which call C code directly and don't have an associated environment.

Closures are useful for making function factories, and are one way to manage mutable state in R.

Function factories

A function factory is a factory for making new functions. We've already seen two examples of function factories, `missing_fixer()` and `power()`. You call it with arguments that describe the desired actions, and it returns a function that will do the work for you. For `missing_fixer()` and `power()`, there's not much benefit in using a function factory instead of a single function with multiple arguments. Function factories are most useful when:

- The different levels are more complex, with multiple arguments and complicated bodies.
- Some work only needs to be done once, when the function is generated.

Function factories are particularly well suited to maximum likelihood problems, and you'll see a more compelling use of them in mathematical functionals.

Mutable state

Having variables at two levels allows you to maintain state across function invocations. This is possible because while the execution environment is refreshed every time, the enclosing environment is constant. The key to managing variables at different levels is the double arrow assignment operator (`<<-`). Unlike the usual single arrow assignment (`<-`) that always assigns in the current environment, the double arrow operator will keep looking up the chain of parent environments until it finds a matching name. (Binding names to values has more details on how it works.)

Together, a static parent environment and `<<-` make it possible to maintain state across function calls. The following example shows a counter that records how many times a function has been called. Each time `new_counter` is run, it creates an environment, initialises the counter `i` in this environment, and then creates a new function.

```
new_counter <- function() {
  i <- 0
  function() {
    i <<- i + 1
    i
  }
}
```

The new function is a closure, and its enclosing environment is the environment created when `new_counter()` is run. Ordinarily, function execution environments are temporary, but a closure maintains access to the environment in which it was created. In the example below, closures `counter_one()` and `counter_two()` each get their own enclosing environments when run, so they can maintain different counts.

```
counter_one <- new_counter()
counter_two <- new_counter()

counter_one()
#> [1] 1
counter_one()
```

```
#> [1] 2
counter_two()
#> [1] 1
```

The counters get around the “fresh start” limitation by not modifying variables in their local environment. Since the changes are made in the unchanging parent (or enclosing) environment, they are preserved across function calls.

What happens if you don’t use a closure? What happens if you use `<-` instead of`<<-`? Make predictions about what will happen if you replace `new_counter()` with the variants below, then run the code and check your predictions.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

Modifying values in a parent environment is an important technique because it is one way to generate “mutable state” in R. Mutable state is normally hard because every time it looks like you’re modifying an object, you’re actually creating and then modifying a copy. However, if you do need mutable objects and your code is not very simple, it’s usually better to use reference classes, as described in RC.

The power of closures is tightly coupled with the more advanced ideas in functionalsand function operators. You’ll see many more closures in those two chapters. The following section discusses the third technique of functional programming in R: the ability to store functions in a list.