To understand computations in [R](https://www.r-project.org/), two slogans are helpful:

* Everything that exists is an object.
* Everything that happens is a function call.

John Chambers

In R, the “[” array access operator is a function call. And it is one a user can re-bind to the new effect of their own choosing.

Let’s see what sort of mischief we can get into using this capability.

Yeah, yeah, but your scientists were so preoccupied with whether or not they could that they didn’t stop to think if they should.

Jurassic Park (1993) – Jeff Goldblum as Dr. Ian Malcolm

How about defining a new [-based function call notation? The ideas is: we could write sin[5] in place of sin(5), thus unifying the notations for function call and array access. Some languages do in fact have unified function call and array access (though often using “(” for both). Examples languages include [Fortran](https://en.wikipedia.org/wiki/Fortran) and [Matlab](https://en.wikipedia.org/wiki/MATLAB).

Let’s add R to the list of such languages. We could define the [ to have either R-traditional [lazy](https://en.wikipedia.org/wiki/Lazy_evaluation) argument semantics.

# lazy argument version

`[` <- function(x, ...) {

args <- as.list(substitute(alist(...)))

args <- do.call(base::`[`, args = list(args, -1))

if(is.function(x)) {

return(do.call(x, args = args))

}

return(do.call(base::`[`, args = c(list(x), args)))

}

Or we could define the [ to have [eager](https://en.wikipedia.org/wiki/Eager_evaluation) argument semantics.

# eager argument version

`[` <- function(x, ...) {

args <- list(...)

if(is.function(x)) {

return(do.call(x, args = args))

}

return(do.call(base::`[`, args = c(list(x), args)))

}

Let’s try the eager version.

sin[5]

#> [1] -0.9589243

c(10,20)[2]

#> [1] 20

c(1,2)[-2]

#> [1] 1

d = data.frame(x= 1:5, y= 2)

d[2, 'y', drop = FALSE]

#> y

#> 2 2

paste0['1', 'c']

#> [1] "1c"

One of the advantages of eager evaluation is: if you know a function is in fact going to use all if its arguments, it often makes sense to compute them all ahead of time. For example: we don’t want a function that runs an expensive step on its first argument to then error-out due to issues that could have been addressed in its second argument.

Notice below how with lazy evaluation it takes 100 seconds to notice the second argument to f(,) is bad. With eager evaluation we detect this instantly.

f <- function(v1, v2) {

Sys.sleep(v1) # simulate expensive step

v2 # oops, inexpensive next step fails

}

date()

#> [1] "Wed Oct 2 11:14:06 2019"

f(100, stop())

#> Error in f(100, stop()):

date()

#> [1] "Wed Oct 2 11:15:46 2019"

With eager evaluation we detect the issue much quicker.

date()

#> [1] "Wed Oct 2 11:15:46 2019"

f[100, stop()]

#> Error in f[100, stop()]:

date()

#> [1] "Wed Oct 2 11:15:46 2019"

Eager languages are more common. Examples include Python, C, C++, Java, and many more. So students are more likely to be already familiar with eager evaluation. Eager languages are also typically considered easier to debug, as it is much easier to infer evaluation order from the source code.

Lazy languages, such as Haskell and R, can save the time wasted in computing values of unused arguments. They also allow users to introduce their own new evaluation control structures, and therefore tend to be very user extensible.