Functional Programming

Dom Owens

23/10/2019

Functional programming is part of the declarative paradigm. Within this, we consider a program to be a sequence of mathematical functions. This corresponds well to the logical form of a program (consider CT Thesis).

First-class functions

We require these three criteria for **first class functions**:

- functions can be arguments to other functions
- functions can be returned by functions
- functions can be stored in data structures.

In summary, we require that functions behave like variables.

Let's demonstrate these criteria with base R.

```
sum(rep(sum(1:5), 5)) #sum as argument to sum
```

```
## [1] 75
```

```
inverse <- function(f) { #returns 1/function
  nf <- function(...) {
    return(1/f(...))
  }
}

inverse.sum <- inverse(`+`) #define function giving 1/sum of values
inverse.sum(1, 2)
```

```
## [1] 0.3333333
```

```
l <- list(add1 = function(x){x+1}, add2 = function(x){x+2}) #create lsit of addition functions
l[[1]](0)
```

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

```
l[[2]](0)
```

```
## [1] 2
```

We can specify *anonymous* functions which are not bound to a name; often this can be used when the function is small and used once.

```
integrate(function(x) sin(x) ^ 2, 0, pi) # integrate sin^2 over one period, as anonymous fn
```

```
## 1.570796 with absolute error < 1.7e-14
```

Pure functions

A **pure** function maps arguments to results independently of the machine's state, and has no side effects. These correspond nicely to mathematical functions.

Impure functions, which violate these requirements, are common in data analysis tasks (consider that we use probabilistic structures; RNG for example depends on the system clock by default)

```
#this is pure
add <- function(x){
  s <- 0
  for (i in 1:length(x)) {
    s <- s+ x[i]
  }
  s
}
add(1:10)
```

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

```
#this is impure
rand <- function(x){
  for (i in 1:length(x)) {
  x[i] <- x[i] + rnorm(1)
  }
  x
}
rand(1:10)
```

```
##  [1]  0.6470691  1.1035987  1.8444178  3.5497409  8.4416492  6.6629045
##  [7]  5.2744130  8.7731999 10.0166894  9.6747533
```

```
rand(1:10)
```

```
##  [1] 1.860630 1.286088 3.027539 4.625390 5.162913 6.701334 8.213206
##  [8] 6.703833 9.668737 8.736238
```

```
#These two results are (probably) not the same
```

Closures

When a function is created by another function, the wider environment of the child function is used to resolve free variables. A **closure** can assign a variable to be used later on.

This attaches data to a function, and makes it possible to express a function of multiple variables as multiple functions of a single variable.

See this example, which creates a discrete probability vector for a multinomial distribution with $N = 10$, then calls on this later as an argument to `rmultinom` to generate samples.

```
make_disc_dist <- function(N){
  prob <- rmultinom(1, N, rep(1/N, N))/N #produce probability score vector
  generate <- function(t){rmultinom(t,N,prob)} #generate samples from score vector
  generate(5) #generate 5
}
make_disc_dist(5) #generate 5 length-5 multinomal samples according to a random score vector
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    2
## [3,]    0    0    0    0    0
## [4,]    5    5    5    5    3
## [5,]    0    0    0    0    0
```

```r
make_disc_dist(5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    2    3    4    1    3
## [3,]    1    1    0    0    1
## [4,]    0    0    0    0    0
## [5,]    2    1    1    4    1
```

Lazy Evaluation

R uses **lazy evaluation**, meaning the number of times an expression is evaluated is minimised. See these examples, noting how the function argument is evaluated.

```r
foo <- function(x) {
  return(10)
}
foo(warning("this warning is not shown"))
```

```
## [1] 10
```

```r
bar <- function(x) {
  force(x)
  return(10)
}
bar(warning("this warning is shown"))
```

```
## Warning in force(x): this warning is shown
```

```
## [1] 10
```

However, this doesn't alwayds work as intended. This is demonstrated in the following example, with a *function factory* for producing functions.

```r
make.repeatvec.function <- function(multiple) { #factory for repeatvec fn
  repeatvec <- function(x) rep(x, multiple)
}
multiple <- 2 #set multiple
two_times <- make.repeatvec.function(multiple) #set repeatvec fn
multiple <- 3 #change multiple
two_times("a") #print "a", multiple times
```

```
## [1] "a" "a" "a"
```

This is unexpected; we wanted this to print "a" two times. The `two_times` function has not been reevaluated. We use `force` to make this work as intended.

```r
make.repeatvec.function <- function(multiple) {
  force(multiple) #reevaluate with current value of multiple
  repeatvec <- function(x) rep(x, multiple)
}
multiple <- 2
two_times <- make.repeatvec.function(multiple)
multiple <- 3
two_times("a")
```

```
## [1] "a" "a"
```

This work refers to http://adv-r.had.co.nz/Functional-programming.html