Advanced R programming: solutions 1

Dr Colin Gillespie

November 29, 2013

1 Argument matching

R allows a variety of ways to match function arguments.[1] We didn't cover argument matching in the lecture, so let's try and figure out the rules from the examples below. First we'll create a little function to help

[1] For example, by position, by complete name, or by partial name.

```
arg_explore = function(arg1, rg2, rg3)
    paste("a1, a2, a3 = ", arg1, rg2, rg3)
```

Next we'll create a few examples. Try and predict what's going to happen before calling the functions

One of these examples will raise an error - why?

```
arg_explore(1, 2, 3)
arg_explore(2, 3, arg1 = 1)
arg_explore(2, 3, a = 1)
arg_explore(1, 3, rg = 1)
```

Can you write down a set of rules that R uses when matching arguments?

```
## SOLUTION
## See http://goo.gl/NKsved for the offical document
## To summeriase, matching happens in a three stage pass:
#1. Exact matching on tags
#2. Partial matching on tags.
#3. Positional matching
```

Following on from the above example, can you predict what will happen with

```
plot(type = "l", 1:10, 11:20)
```

and

```
rnorm(mean = 4, 4, n = 5)
```

```
## SOLUTION
#plot(type="l", 1:10, 11:20) is equivilent to
plot(x=1:10, y=11:20, type="l")
#rnorm(mean=4, 4, n=5) is equivilent to
rnorm(n=5, mean=4, sd=4)
```

2 *The ... argument*

A common argument[2] is .... We can explore what happens using the `eval` and `substitute` functions.

[2] Especially when dealing with S3 objects and functions.

```
arg_explore2 = function(arg1 = 5, ...)
    eval(substitute(alist(...)))
```

- What do `alist`, `substitute` and `eval` do?[3]

[3] Hint: the easiest way to figure this out is to alter the `arg_explore2` function, i.e. remove `eval`, then remove `substitute`, etc.

```
## SOLUTION
#1. eval - just evalulats an R expression
#2. substitute - returns the unevaluated expression
#3. alist - Used to parse the arguments
#Look at ?alist, ?eval and ?substitute
#Also, run the examples - example(eval)
```

- Repeat the examples used in `arg_explore`, but include the ... argument.

3 *Variable scope*

Scoping can get tricky. **Before** running the example code below, predict what is going to happen

1. A simple one to get started

```
f = function(x) return(x + 1)
f(10)
```

```
##Nothing strange here. We just get
f(10)

## [1] 11
```

2. A bit more tricky

```
f = function(x) {
    f = function(x) {
        x + 1
    }
    x = x + 1
    return(f(x))
}
f(10)
```

3. More complex

```
f = function(x) {
    f = function(x) {
        f = function(x) {
            x + 1
        }
        x = x + 1
        return(f(x))
    }
    x = x + 1
    return(f(x))
}
f(10)
```

```
## Solution: The easiest way to understand
## is to use print statements
f = function(x) {
    f = function(x) {
        f = function(x) {
            message("f1: = ", x)
            x + 1
        }
        message("f2: = ", x)
        x = x + 1
        return(f(x))
    }
    message("f3: = ", x)
    x = x + 1
    return(f(x))
}
f(10)

## f3:  = 10
## f2:  = 11
## f1:  = 12

## [1] 13
```

4.
```
f = function(x) {
    f = function(x) {
        x = 100
        f = function(x) {
            x + 1
        }
        x = x + 1
        return(f(x))
    }
    x = x + 1
```

```
    return(f(x))
}
f(10)
```

```
## Solution: The easiest way to understand
## is to use print statements as above
```

4 Function closures

Following the examples in the notes, where we created a function closure for the normal and uniform distributions. Create a similar closure for

- the Poisson distribution,[4]

```
poisson = function(lambda) {
    r = function(n = 1) rpois(n, lambda)
    d = function(x, log = FALSE) dpois(x,
        lambda, log = log)
    return(list(r = r, d = d))
}
```

[4] Hint: see `rpois` and `dpois`.

- and the Geometric distribution.[5]

```
geometric = function(prob) {
    r = function(n = 1) rgeom(n, prob)
    d = function(x, log = FALSE) dgeom(x,
        prob, log = log)
    return(list(r = r, d = d))
}
```

[5] Hint: see `rgeom` and `dgeom`.

5 Mutable states

In chapter 2, we created a random number generator where the `state`, was stored between function calls.

- Reproduce the `randu` generator from the notes and make sure that it works as advertised.
- When we initialise the random number generator, the very first state is called the `seed`. Store this variable and create a new function called `get_seed` that will return the initial seed, i.e.

```
r = randu(10)
r$r()

## [1] 0.0003052
```

```
r$get_state()

## [1] 655390

r$get_seed()

## [1] 10
```

```
## Solutions - see below
```

- Create a variable that stores the number of times the generator has been called. You should be able to access this variable with the function `get_num_calls`

```
r = randu(10)
r$get_num_calls()

## [1] 0

r$r()

## [1] 0.0003052

r$r()

## [1] 0.001831

r$get_num_calls()

## [1] 2
```

```
## Solutions
randu = function(seed) {
    state = seed
    calls = 0  #Store the number of calls
    r = function() {
        state <<- (65539 * state)%%2^31
        ## Update the variable outside of this
        ## enviroment
        calls <<- calls + 1
        state/2^31
    }
    set_state = function(initial) state <<- initial
    get_state = function() state
    get_seed = function() seed
    get_num_calls = function() calls
    list(r = r, set_state = set_state, get_state = get_state,
```

```
        get_seed = get_seed, get_num_calls = get_num_calls)
}
r = randu(10)
r$r()

## [1] 0.0003052

r$get_state()

## [1] 655390

r$get_seed()

## [1] 10
```

Solutions

Solutions are contained within the course package

```
library("nclRadvanced")
vignette("solutions1", package = "nclRadvanced")
```