Advanced R programming: practical 1

Dr Colin Gillespie

November 25, 2014

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?

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)
```

2 Functions as first class objects

Suppose we have a function that performs a statistical analysis

```
## Use regression as an example
stat_ana = function(x, y) {
  lm(y ~ x)
}
```

However, we want to alter the input data set using different transformations[2]. In particular, we want the ability to pass arbaritary transformation functions to `stat_ana`.

[2] For example, the log transformation.

- Add an argument `trans` to the `stat_ana` function. This argument should have a default value of `NULL`.

- Using `is.function` to test whether a function has been passed to `trans`, transform the vectors `x` and `y` when appropriate. For example,

```
stat_ana(x, y, trans=log)
```

  would take log's of `x` and `y`.

- Allow the `trans` argument to take character arguments in additional to function arguments. For example, if we used `trans = 'normalise'`, then we would normalise the data[3].

[3] Subtract the mean and divide by the standard deviation.

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)
```

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)
```

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)
```

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]
- and the Geometric distribution.[5]

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

[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
```

- 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

Solutions are contained within the course package

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