

Advanced R programming: solutions 2

Dr Colin Gillespie

January 1, 2014

1 S3 objects

1. Following the cohort example in the notes, suppose we want to create a mean method.

- List all S3 methods associated with the mean function.

```
methods("mean")  
  
## [1] mean.Date      mean.default mean.difftime  
## [4] mean.POSIXct    mean.POSIXlt
```

- Examine the source code of mean.default.

```
body("mean.default")
```

- What are the arguments of mean.default?

```
args("mean")  
  
## function (x, ...)  
## NULL
```

- Create a function called mean.cohort that returns a vector containing the mean weight and mean height.¹

```
mean.cohort = function(x, ...) {  
  m1 = mean(x$details[, 1], ...)  
  m2 = mean(x$details[, 2], ...)  
  return(c(m1, m2))  
}
```

¹ Ensure that you can pass in the standard mean arguments, i.e. na.rm.

2. Let's now make a similar function for the standard deviation

- Look at the arguments of the sd function.
- Create an function call sd.cohort that returns a vector containing the weight and height standard deviation.²
- Create a default sd function. Look at cor.default in the notes for a hint.

```
sd = function(x, ...) UseMethod("sd")  
sd.default = function(x, ...) stats::sd(x, ...)  
sd.cohort = function(x, ...) {  
  s1 = sd(x$details[, 1], ...)  
  s2 = sd(x$details[, 2], ...)  
  return(c(s1, s2))  
}
```

² Ensure that you can pass in the standard sd arguments, i.e. na.rm.

2 S4 objects

- Following the Cohort example in the notes, suppose we want to make a generic for the mean function.

I've intentionally mirrored the functions from section 1 of this practical to highlight the differences.

- Using the `isGeneric` function, determine if the mean function is an S4 generic. If not, use `setGeneric` to create an S4 generic.

```
isGeneric("mean")
## [1] FALSE

setGeneric("mean")
## [1] "mean"
```

- Using `setMethod`, create a mean method for the Cohort class.³

³ Be careful to match the arguments.

```
setMethod("mean", signature = c("Cohort"), definition = function(x,
  ...) {
  m1 = mean(x@details[, 1], ...)
  m2 = mean(x@details[, 2], ...)
  return(c(m1, m2))
})
## [1] "mean"
```

- Repeat the above steps for the sd function.

```
isGeneric("sd")
## [1] FALSE

setGeneric("sd")
## [1] "sd"

setMethod("sd", signature = c("Cohort"), definition = function(x,
  na.rm = FALSE) {
  m1 = sd(x@details[, 1], na.rm = na.rm)
  m2 = sd(x@details[, 2], na.rm = na.rm)
  return(c(m1, m2))
})
## [1] "sd"
```

3 Reference classes

The example in the notes created a random number generator using a reference class.

- Reproduce the `randu` generator from the notes and make sure that it works as advertised.⁴

⁴ The reference class version, not the function closure generator.

- 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(calls = 0, seed = 10, state = 10)
r$r()

## [1] 0.0003052

r$get_state()

## [1] 655390

r$get_seed()

## [1] 10
```

Reference classes also have an initialise method - that way we would only specify the seed and would then initialise the other variables. I'll give you an example in the solutions.

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(calls = 0, seed = 10, state = 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("solutions2", package = "nclRadvanced")
```