R from a programmer's perspective

Solutions to the exercises within an R course held by M. Göker at the DSMZ, 11/05/2012 & 25/05/2012.

Slightly improved version, 10/09/2012.

R as programming language

1. Function serve as subprograms, packages as modules. R has lexical scope.
2. Functions can be created by functions and can be passed to functions. Data structures and functions can be converted to each other.
3. There is no referential transparency.
4. E.g. `x <- 5; x <- "abc"` would be sufficient to show that.
5. E.g. that `5 + "5"` results in an error would be sufficient to show that.
6. Important examples are the automated coercion when `c()` is applied to vector types other than lists and when `==` is used; also, numeric values can be used in `if`, `else` and `while` statements.
7. E.g. that `1:5 + 1` is possible and results in a vector of length 5 would be sufficient to demonstrate vectorization. Keep in mind that the order of the two vectors does not matter. The easy cases are if the length of one vector is an integer multiple of the length of the other one, with the two frequent trivial cases of length-one vectors and equal-length vectors. Make sure you understand the behaviour in the case of zero-length vectors.
8. Logical or numeric vectors to be used in `if`, `else` and `while` statements need not have length 1, even though this would be the only sensible value.
9. These exercises show that `==` and `identical()` differ regarding vectorization and whether they do automated coercion. The solutions are, in order:
   - `rep.int(TRUE, 26L)`
   - `rep.int(TRUE, 26L)`
   - `logical()`
   - `rep.int(FALSE, 26L)`
   - `c(TRUE, NA, rep.int(c(FALSE, NA), 12L))`
   - `TRUE`
   - `FALSE`
   - `TRUE`
   - `TRUE`
   - `TRUE`

- `FALSE`
- `c(FALSE, TRUE, FALSE)`

10. Enter `?Reserved` and `?Syntax` at the R prompt and study this part of the documentation.

R's basic types

1. raw > character > complex > numeric > integer > logical
2. It can have zero length, and it can contain `NA` values.
3. R tries to hide the difference between integers and doubles from the user. Also, because these are implicit classes, method dispatch is C-internal. Implicitly, numeric values have the class `c('double', 'numeric')`; integer values have the class `c('integer', 'numeric')`.
4. Data frames have an explicit class, set via the according attribute.
5. Lists can contain lists, functions can contain functions (in contrast to, say, C).
6. `m <- matrix(c(T, F, T, F), ncol = 2); mode(m) <- "numeric"`

Writing functions in R

1. Function with no arguments

This is the simplest solution, because we did not request a certain function body.

```
fun_1 <- function() {}
```

2. Function with arbitrary number of arguments

The first, simple solution introduces the important `...` reserved word; again the function body is arbitrary. The second solution is more in agreement with the literature (Douglas-Adams memorial solution).

```
fun_2 <- function(...) {}

the_answer <- function(...) {

  42L # this is the right answer, irrespective of the arguments!

}
```

3. Function accepting 1 or 2 arguments

The simple first solution just ignores the 2nd argument. The next one is a solution with a default argument. The third one checks whether or not the 2nd has been passed (this is possible because of R's lazy-evaluation mechanism).

```
fun_3a <- function(a, b) {

  a
}

fun_3b <- function(a, b = 3) {

  c(a, b) # arbitrary, but should need both arguments

}

fun_3c <- function(a, b) {

  if (missing(b))

    b <- 3

  c(a, b) # arbitrary, but should need both arguments

}
```

4. Function with 1-2 arguments, 2nd needed dependent on 1st

This function also demonstrates that `if` and `else` have return values.

```
fun_4 <- function(a, b) {

  if (is.null(a)) # the condition is actually arbitrary but must use 'a'

    b

  else

    a

}
```

5. Function that returns last argument

By naming 'x', we ensure that at least one argument must is given; hence we avoid indexing with 0. Note that built-in functions such as `list()` that accept arbitrary numbers of arguments themselves ease the use of `...`.

```
fun_5 <- function(x, ...) {

  list(x, ...)[[nargs()]]

}
```

6. Function that returns randomly selected argument

Like function #5, only some computation on nargs() is needed. The solution with `runif()` works because when indexing with numeric values, they are truncated towards 0. The second, even more elegant solution has been suggested by Jürgen Tomasch; I have just inserted `sample.int()` instead of `sample()`.

```
fun_6a <- function(x, ...) {

  list(x, ...)[[runif(1L, 1L, nargs() + 1L)]]
}

fun_6b <- function(x, ...) {

  list(x, ...)[[sample.int(nargs(), 1L)]]

}
```

7. Function that returns its n-th argument

The following function returns the 3rd argument (the number is arbitrary) and crashes with fewer arguments.

```
fun_7 <- function(...) {

  ..3

}
```

8. Function that returns n-th argument, n being selected by 1st argument

The first solution avoids creating a list from `...` but it uses the ugly eval() construct. Regarding the third solution, keep in mind that `switch()` can be passed a numeric first argument; in contrast to the first two functions, it but does not result in out-of-range errors (it silently returns NULL in such cases, which might or might not be what you want). All three functions crash with an invalid first argument.

```
fun_8a <- function(...) {

  eval(parse(text = sprintf("..%i", ..1)))

}

fun_8b <- function(...) {

  items <- list(...)

  items[-1L][[items[[1L]]]]

}

fun8c <- function(...) {

  switch(...)

}
```

9. Function whose last argument must be named

Here we must name 'x' because otherwise it is treated as part of '...'.

```
fun_9 <- function(..., x) {

  x
}
```

10. Function that calculates mean, min, or max

The first solution introduces an important use of `switch()`. It would here be the best solution but involves the repetition of the function names. The second solution uses `do.call()`, which makes this rather elegant. An enhancement would be to deal with the 'na.rm' argument of each of the three functions.

```
fun_10a <- function(x, what = c("mean", "min", "max")) {
  switch(match.arg(what),
    mean = mean(x),
    min = min(x),
    max = max(x)
  )
}
fun_10b <- function(x, what = c("mean", "min", "max")) {
  do.call(match.arg(what), list(x))
}
```

11. Function that calculates a function's arity

We here determine the arity using `formals()`, which yields all formal arguments as pairlist (which behaves roughly like a named list). Note the difference between `formals()` and `nargs()`!

```
fun_11 <- function(fun) {
  formal.names <- names(formals(fun))
  if ("..." %in% formal.names)
    Inf
  else
    length(formal.names)
}
```

12. Function that returns its call

Here `format()` works much nicer than `as.character()`. The knowledge of `match.call()` is crucial.

```
fun_12 <- function(...) {
  format(match.call())
}
```

13. Function that returns itself

Here we use `as.character()` combined with `match.call()`. The elegance here lies in the fact that we could rename the function at will but it would still return itself. Somewhat less safe would be `get()` instead of `match.fun()` unless one uses the `mode = "function"` argument. The second function uses the built-in solution.

```
fun_13a <- function(...) {

  match.fun(as.character(match.call())[1L])

}

fun_13b <- function(...) {

  sys.function()

}
```

14. Function that creates a counter

This example demonstrates how closures can be created in R. You might want to create a function using `fun_14()` and then study this function's environment, e.g. `counter <- fun_14(); as.list(environment(counter))`.

```
fun_14 <- function(cnt = 1L) {

  cnt <- cnt - 1L

  function(incr = 1L) (cnt <<- cnt + incr)

}
```

15. Function that reverts the arguments of another one

The first function is more elegant because it uses the predefined getter and setter functions for formal arguments. The second, more complicated version shows that R is homoiconic insofar as data structures (lists) and functions can easily be converted to each other.

```
fun_15a <- function(fun) {

  formals(fun) <- rev(formals(fun))

  fun

}

fun_15b <- function(fun) {

  if (!is.function(fun))

    stop("'fun' must be a function")

  fun <- as.list(fun)
  fl <- length(fun) # the last list element holds the function body

  fun <- c(rev(fun[-fl]), fun[fl])

  as.function(fun)

}
```

Error handling

1. `unpercent()` with argument checking

Solutions using either `stop()` or `stopifnot()`. Note the use of `||` for short-circuit evaluation (instead of `|`) and the use of `any()` in conjunction with `na.rm = TRUE` because `if` crashes if it gets passed a `NA` value.

```
unpercent <- function(x) {

  if (!is.numeric(x))

    stop("'x' must be numeric")

  if (any(y < 0, na.rm = TRUE) || any(y > 100, na.rm = TRUE))

    warning("'x' should be between 0 and 100")

  x / 100

}

unpercent2 <- function(x) {

  stopifnot(is.numeric(x))

  if (any(y < 0, na.rm = TRUE) || any(y > 100, na.rm = TRUE))

    warning("'x' should be between 0 and 100")

  x / 100

}
```

2. Turn warnings to errors

Note the explicit naming of all explicitly passed arguments because we also pass `...` to `tryCatch()`. Improvements would include passing other arguments to `stop()`, such as `call.`, or allowing other error-message texts.

```
must <- function(expr, ...) {

  tryCatch(expr = expr,

    warning = function(w) stop(conditionMessage(w)), ...)

}
```

3. Turn errors into their messages

```
taste <- function(expr, ...) {
  tryCatch(expr = expr, error = conditionMessage, ...)
}
```

4. Turn warnings into messages

Note the use of on.exit() assuring that global options are reset.

```
relaxed <- function(expr) {
  ops <- options(warn = -1)
  on.exit(options(ops))
  withCallingHandlers(expr, warning = function(w) {
    message(conditionMessage(w))
    invokeRestart("muffleWarning")
  })
}
```

Attributes

1. It is not a syntactical name.
2. `"George W. Bush"`
3. `attr(x, "attribute.name") <- NULL; attributes(x) <- NULL`

1. Getter and setter for an attribute

```
`feature<-` <- function(x, value) {
  attr(x, "feature") <- value
  x
}
feature <- function(x) {
  attr(x, "feature")
}
```

2. Attributes of a matrix

The attributes are called “dim” and “dimnames”.

3. Attributes of a data frame

The attributes are called “class”, “names” and “rownames”.

4. Turning a matrix to a vector

`attr(m, "dim") <- NULL` would be sufficient because it removes the “dim” attribute. The attribute “dimnames” would automatically be deleted because it made no sense any more.

5. Setter function using `structure()`

```
`author<-` <- function(x, value) {
  structure(.Data = x, author = value)
}
```

6. Getter and setter via metaprogramming

```
set_getter_and_setter <- function(name, pos = 1L, ...) {
  assign(x = name, value = function(x) attr(x, name), pos = pos, ...)
  assign(x = sprintf("%s<-", name), value = function(x, value) {
    attr(x, name) <- value
    x
  }, pos = pos, ...)
  invisible(name) # return value is arbitrary
}
```

Note the use of `invisible()`: because we call the function for its side effect, we do not want its return value to be printed to the screen. (By the way, putting the function call in parentheses would cause the return value to be printed irrespective of `invisible()`.) An improvement of this function would be to vectorize it regarding `name` and perhaps `pos`:

```
set_getter_and_setter <- function(names, pos = 1L, ...) {
  getter <- function(name) function(x) attr(x, name)
  setter <- function(name) function(x, value) {
    attr(x, name) <- value
    x
  }
  more <- list(...)
  mapply(assign, names, lapply(names, getter), pos, MoreArgs = more)
  mapply(assign, sprintf("%s<-", names), lapply(names, setter), pos,
    MoreArgs = more)
  invisible(names)
}
```

Implementing a generic "Hello world" function and its methods

1. Hello-world method for numeric vectors

```
hello_world.numeric <- function(x) {
  # rep.int() does the argument checking for us
  print(rep.int("Hello world!", x))
}
```

2. Hello-world method for data frames

```
hello_world.data.frame <- function(x) {
  print(sprintf(
    "Hello world, I am a data frame with %i rows and %i columns!",
    nrow(x), ncol(x)))
}
```

3. Hello-world method for factors

```
hello_world.factor <- function(x) {
  hello_world(as.character(x))
}
```

Implementing a "Hello world" class

1. What do `data.frame()`, `factor()`, `table()` and `glm()` have in common?

As usual for S3 constructor functions, they first built a list or vector object together and then add an explicit class to override the default behaviour of lists and vectors. This is the only way in S3 to guarantee that classes have the content they are expected to contain: by using only the constructor functions dedicated for this purpose.

2. How does “hello-worldifying” objects change their behaviour?

The “hello.world” `print()` method is selected, and if the variable name of such an object is entered at the R prompt, “Hello world!” is printed.

3. What happened if `class()` replaces `oldClass()`?

Implicit classes would be set explicitly, even though this is unnecessary and could be confusing.

4. What happened if `insert_class()` would insert at the end?

The “hello.world” `print()` method would not be the first choice any more, and the objects' printing behaviour not necessarily changed.

5. Remove a class name from an S3 object

```
delete_class <- function(x, klass) {
  if (isS4(x))
    stop("this function is not intended for S4 objects")
  if (length(klass <- as.character(klass)) < 1L)
    stop("empty 'klass' argument")
  # setdiff() would be more invasive
  class(x) <- oldClass(x)[!oldClass(x) %in% klass]
  x
}
unmake_hello_world <- function(x) {
  delete_class(x, "hello_world")
}
```

6. `unclass()` vs. `delete_class()`

In contrast to `delete_class()`, `unclass()` removes all explicit classes. If applied to a data frame, a list is returned, with the data-frame columns as elements.

7. `arity()` and `unpercent()` functions as S3 method

The function body can be simplified because automated method dispatch makes certain checks unnecessary. Note, however, that the previous `unpercent()` function would also work with matrices and array; hence, we need to take care of them here, too. Note the trick with `[]`.

```
arity <- function(fun) UseMethod("arity")
```

```
arity.function <- fun_11
unpercent <- function(x) UseMethod("unpercent")
unpercent.numeric <- function(x) {
  if (any(y < 0, na.rm = TRUE) || any(y > 100, na.rm = TRUE))
    warning("'x' should be between 0 and 100")
  x / 100
}
unpercent.matrix <- function(x) {
  x[] <- unpercent(as.vector(x))
  x
}
unpercent.array <- function(x) {
  x[] <- unpercent(as.vector(x))
  x
}
```

8. `asqr()` in yarp

`asqr()` apparently only makes sense for numeric data. The default method converts its input via `as.numeric()`. The matrix method ensures that the dimensions remain the same. The `...` operator ensures that methods can define additional arguments.

8. `box_cox_fun()` and `box_cox()` in yarp

`box_cox_fun()` creates a function from a numeric value `y` that would conduct the conversion defined by `y`. `box_cox()` uses such functions to conduct the conversions. If several `y` are given, conversions are done for all of them, using lists of functions as intermediate product.

9. `box_cox_fun()` compared to the "counter" example

Like the counter functions, the result of `box_cox_fun()` is a function with its own environment, used for storing either the state of the counter or the `y` value for the box-cox transformation. These are examples for so-called closures in R.

S3 group generics

1. Effect of helloworldizing a vector

One should get the annoying messages each time one of the functions belonging to the group generic is called.

2. `Ops` group generic for “hello.word” class

```
Ops.hello.world <- function(x, ...) { # example code #11

  message(sprintf("Hello world, let's compute '%s'!", .Generic))

  NextMethod()

}
```

The effect should be the same as for the `Summary` group generic but just affect other functions.

Creating R packages

1. Use of the DESCRIPTION file

DESCRIPTION is the master file of a package. It clarifies the title, description, dependencies, and collating order of the package's R files. Its main entries are copied to the 1st page of the PDF manual.

2. Use of the NAMESPACE file

The NAMESPACE file explains which functions to export from the package, and which of them are S3 or S4 methods. All other functions remain package-internal and cannot normally called by a user, even after loading the package with `library()`.

3. Documentation generation from the Rd file of `asqr()`

The *\name*, *\alias* and *\keyword* entries are for generating the index of the manual. All other commands directly compose the manual page for the function.

3. Generation of the Rd file for `asqr()`

The `@family` directive creates the *\seealso* links (to all other functions of the family). `@export` is for creating the NAMESPACE entry (see below). `@param` and `@inheritParams` go to *\arguments*. The remaining directives have direct counterparts in the Rd files. The first lines yields *\title*, the next paragraph yields *\description*.

3. Generation of the NAMESPACE entries for `asqr()`

`@export` and `@method` are necessary here. `@method` causes `S3method()` to be written, but only in conjunction with `@export`. The generic functions need `export()` in the NAMESPACE file.

3. Hidden helper functions

`@keywords` internal causes such functions not to be documented. They are not exported because `@export` is missing.

3. Re-generating the documentation

Within the “docu.R” script, the call of `roxygenize()` causes the Rd files to be generated form the R files. The call of `update_pack_desc()` updates the DESCRIPTION file. Note the first line, a Shebang line, which enables one to call the script directly from the command line, without starting an interactive R session.

4. Running R CMD check

The directory should contain the files “00check.log”, “00install.out”, “Rdlatex.log”, “yarp-Ex.R”, “yarp-Ex.Rout”, “yarp-Ex.pdf”, “yarp-manual.log”, “yarp-manual.pdf” and the subdirectory “yarp”, a copy of the package. “yarp-Ex.Rout” contains the results from running the examples. “00check.log” contains what was also printed to the screen when running R CMD check.

5. Additional files in the opm package

The “data” subdirectory contains example datasets coming with the package to be loaded using `data()`. “NEWS” lists the major changes in each novel version. The content of the “inst” subdirectory is copied without modification to the installation directory when running R CMD INSTALL.