R from a programmer's perspective

Accompanying manual for an R course held by M. Göker at the DSMZ, 11/05/2012 & 25/05/2012.

Slightly improved version, 10/09/2012.

Introduction

The purpose of this course is to cover aspects of R programming that are either unlikely to be covered elsewhere or likely to be surprising for programmers who have worked with other languages. The course thus tries not be comprehensive but sort of complementary to other sources of information. Also, the material needed to by compiled in short time and perhaps suffers from important omissions. For the same reason, potential participants should not expect a fully fleshed out presentation but a combination of a text-only document (this one) with example code comprising the solutions of the exercises.

The topics covered include R's general features as a programming language, a recapitulation of R's type system, advanced coding of functions, error handling, the use of attributes in R, object-oriented programming in the S3 system, and constructing R packages (in this order).

The expected audience comprises R users whose own code largely consists of self-written functions, as well as programmers who are fluent in other languages and have some experience with R. Interactive users of R without programming experience elsewhere are unlikely to benefit from this course because quite a few programming skills cannot be covered here but have to be presupposed. We also need to limit the number of participants because most topics will be covered by discussing questions and exercises. Some exclusiveness is thus needed here. The participants, however, are invited to further distribute the results of the course (if any) via the “R club”.

The appendix of this course manual contains the first version of a DSMZ R style guide. The code examples in this document also serve as examples for the application of this style guide.

Conventions used in this document

In the following, “knowledge questions” are questions that you should attempt to answer without empirically assessing them by typing code in the R interpreter. In contrast, the “exercises” request you to write and try R code. The exercises are usually sorted increasingly in terms of their difficulty. Important keywords are written in *italics*. Code within this document is typed in `monospace` and formatted according to the DSMZ R style guide in the appendix. Package names are written in **bold face**. Curses and offences have been [redacted].

Further requirements

In addition to this document, you will need:

- Access to a computer with a running R version.
- An editor for R code, preferably one with appropriate syntax highlighting.
- The package **roxygen2** from CRAN installed into this R version.

- The package **pkgutils** (eventually available at CRAN) installed and the contained “docu.R” script made executable and linked from a $PATH directory (or placed wherever you will find it again). Using it (*not* from R, but from the command line) should be straightforward on Unix-derived systems, but Windows users might additionally need Rtools (http://cran.r-project.org/bin/windows/Rtools).
- The package **yarp** (received from the course instructor) *not* installed but available for unpacking and studying.
- Optionally also the package **opm** from CRAN *not* installed but available for unpacking and studying.

R as programming language

The purpose of this section is to clarify aspects of R programming by revisiting some terms from programming theory (the *object-oriented programming* paradigm will be treated below).

Procedural programming emphasizes modularity. Programs should be composed of subprograms or modules acting as independently as possible. This efficiently increases readability, testability, and reliability, as well as the chances for code reuse. Sensible scoping rules make writing modular code easier.

Functional programming (as opposed to *imperative programming*) is a paradigm that treats functions like mathematical functions by avoiding any side effects. Programs should thus be referentially transparent, i.e. independent of their state at a certain time point. As a consequence, in pure functional programming languages variables can be assigned only once, loops are replaced by recursion, and side effects are typically restricted to IO operations. Further, functions should also be treated as “first-class citizens”; that is, functions should be able to modify and return functions and get functions passed as arguments.

Type systems are relevant for programming because the affect both the ease with which programs can be written as well as how stringently the program is checked directly during interpretation or compilation. In contrast to *statically typed* languages, *dynamically typed* ones allow variables and function arguments to be assigned not only several times but also with contents of distinct types. (Remember that in this context, “type” refers to classes, explicitly set types or basic types such as floats, integers, character strings etc.) *Strong typing* implies that operations involving variables of the wrong types result in an error, whereas *weak typing* characterizes programming languages that attempt to conduct implicit coercions (type casts) in such cases.

Array programming is a feature that allows the programmer to apply operations to entire array-like collections of values at once. An advantage is that explicit looping can be avoided, yielding terser code. Particularly R heavily relies on array programming. (The fact that, unfortunately, arrays are called “vectors” in R and multi-dimensional matrices are called “arrays”, does not matter here. By the way, R “lists” are not lists either.) To fully exploit R's specific capabilities, users must fully understand its approach to array programming. As an interpreted language, using “vectorization” in R usually also leads to speed gains because the relevant looping is then done in the interpreter's underlying compiled code. Vectorization is also highly relevant for R's powerful indexing and subsetting capabilities. A downside of vectorization is that it makes flow control more complicated than in other scripting languages. Actually, all operations for which scalars are required might need more checking in R than in language in which the fact that an object is a scalar can be inferred from its class (which one cannot do in R because of its vectorization approach).

Some aspects of the way R is implemented should also be recalled. R is an interpreted, garbage-collected scripting language which recently included support for byte-code compilation. The R interpreter is a free-software re-implementation of the S programming language and written in C (using some libraries written in Fortran). It internally works as an interpreter for the Scheme programming language (which is a dialect of Lisp). As such, R is largely *homoiconic*. On top of that, R relies on a user-visible Algol-like *syntax* which in its use of parentheses, brackets and curly braces much resembles C but contains some significant deviations (some of which are rather ill-chosen in my view). As belonging to the Algol family of languages, R contains a number of well-known *reserved words*.

Knowledge questions

1. How does R enable procedural programming (with respect to subprograms, modules and scoping)?
2. Which features of R are borrowed from functional programming?
3. Demonstrate that R is not a pure functional-programming language.
4. Demonstrate that R is dynamically typed.
5. Demonstrate that R is strongly typed (in many aspects).
6. Provide at least two counterexamples in which R behaves like a weakly typed language.
7. Demonstrate that R is an array-programming language. Can you fully explain R's “recycling rules”?
8. In which way does vectorization make flow control more difficult?
9. What is the outcome of the following commands (remember that `letters` and `LETTERS` are character vectors of length 26 that contain the characters of the alphabet in lower case and upper case, respectively, defined as constants in the **base** package)?
   - `letters == letters`
   - `letters == letters[TRUE]`
   - `letters == letters[FALSE]`
   - `letters == LETTERS`
   - `letters == c("a", NA)`
   - `identical(letters, letters)`
   - `identical(letters, LETTERS)`
   - `length(letters) == length(LETTERS)`
   - `identical(length(letters), length(LETTERS))`
   - `length(letters) == 26`
   - `identical(length(letters), 26)`
   - `c("3", "2", "1") == 1:3`

10. Which of the following entries are reserved words in R? Which ones are not reserved words but nevertheless predefined in R? If so, what are they (e.g. constants, operators, functions etc.)? Which of the undefined ones *could* be used as names of variables?

```
R               T               TRUE            MAYBE           FALSE

else            elif            elsif           ifelse          if

unless          useless         goto            comefrom        repeat

exit            break           quit            bye             stop

while           do              undo            switch          case

def             procedure       function        in              out

and             or              xor             which           that

formals         for             fork            formula         fortknox

NaN             NA              NAJA            NIL             NULL

^               >               <               ->              <-

=>              >=              ..              ..3             ...

!               ?               .               :               :-)
```

R's basic types

Table 1 provided an overview of the basic types in R. All of them have a specific return value of the `class()` function but are so-called "implicit classes". `is.object()` returns `FALSE` for values from these classes (more information on the distinction between implicit and explicit classes is given below). In addition to construction functions such as `character()`, `logical()` etc., all listed classes have a coercion functions like `as.character()`, `as.logical()` etc.; even an `as.null()` function is present (with the obvious return value). Furthermore, all listed classes come with type-checking functions such as `is.character()`, `is.logical()` etc., but keep in mind that these functions are not equivalent to whether on object belongs the respective class.

Table 1. List of implicit classes built into R, and their main features depicted as the relationships between `class()`, `mode()`, `storage.mode()` and `typeof()`.

Return value when applying...						**Eponymous construction function present?**	**NA value**
`is.atomic`	**`is.vector`**	**`class`**	**`mode`**	**`storage.mode`**	**`typeof`**		
`TRUE`	`FALSE`	`"NULL"`	`"NULL"`	`"NULL"`	`"NULL"`	no	[none]
`TRUE`	`TRUE`	`"raw"`	`"raw"`	`"raw"`	`"raw"`	yes	[none]

TRUE	TRUE	"logical"	"logical"	"logical"	"logical"	yes	NA
TRUE	TRUE	"integer"	"numeric"	"integer"	"integer"	yes	NA_integer_
TRUE	TRUE	"numeric"	"numeric"	"double"	"double"	yes	NA_real_
TRUE	TRUE	"complex"	"complex"	"complex"	"complex"	yes	NA_complex_
TRUE	TRUE	"character"	"character"	"character"	"character"	yes	NA_character_
FALSE	TRUE	"list"	"list"	"list"	"list"	yes	[none]
TRUE	FALSE	"matrix"	[varying]	[varying]	[varying]	yes	[varying]
TRUE	FALSE	"array"	[varying]	[varying]	[varying]	yes	[varying]
FALSE	FALSE	"function"	"function"	"function"	"closure"	yes	[none]
FALSE	TRUE	"expression"	"expression"	"expression"	"expression"	yes	[none]
FALSE	FALSE	"call"	"call"	"language"	"language"	yes	[none]
FALSE	FALSE	"name"	"name"	"symbol"	"symbol"	yes	[none]

Knowledge questions

1. Assume you use `c()` to combine two distinct ones of the vector types list in Table 1. Which class will result from the coercion? Clarify this for all pairs by defining an order of coercion.
2. Provide two reasons why identifying an object as a vector-like one with a certain class does not guarantee that it contains any information.
3. Have a look at the anomalies in Table 1 for the vectors that hold numbers. Where might these anomalies come from?
4. Why are data frames missing from Table 1?
5. A function with an opposite behaviour of `is.atomic()` for almost all classes listed above is `is.recursive()`. Explain in which way the classes for which `is.atomic()` returns `FALSE` (except “call” and “name”, which you can safely ignore) are recursive data structures.
6. How can one convert a logical matrix into a numeric matrix?

Writing functions in R

We will here focus on R's powerful argument-processing capabilities and on the reflection tools that are provided for R functions. Other aspects, such as algorithms, are not specific to functions. Generic functions as used for object-orientation in R are treated below. R's processing of function arguments is powerful as it was designed from the very beginning for using named arguments, arguments with defaults and arbitrary numbers of arguments. Moreover, lazy-evaluation mechanisms enable a programmer to apply some interesting constructs. For instance, the following function:

```
divide <- function(x, y = mean(x)) { # example code #1
```

```
  x / y
}
```

...uses as default second argument an expression that refers to the first argument. This works because `mean(x)` is not evaluated before `x / y` is computed, and never evaluated if the default `y` is overwritten. Similarly, it is possible in argument defaults to refer to variables that are not available in the environment but are set within the function body (the only action needed to make this useful is to describe the meaning of this variable in the documentation of the function).

Another topic is the definition of replacement functions, which will be explained in the chapter on R attributes.

Exercises

None of the functions in the these exercises needs to be directly useful for real programming tasks. Unless a certain return value is requested, try to write (or at least conceive) each functions as being independent of a certain return value. Several solutions might exist, but the more elegant the solution, the better. For simplicity, omit any user-friendly error handling.

1. Write a function that accepts no arguments.
2. Write a function that accepts an arbitrary number of arguments.
3. Write a function that accepts one or two arguments but neither less nor more. Then write a function that accepts zero or one arguments but neither less nor more. Do this with and without default arguments.
4. Write a function that accepts a single argument or needs two ones, but never more, depending on the value of the first argument.
5. Write a function that accepts an arbitrary number of arguments, but at least one, and returns the last one.
6. Write a function that accepts an arbitrary number of arguments, but at least one, and returns one of them, selected at random (this is just a refinement of #5).
7. Write a function that expects at least n arguments, where n is an arbitrary positive integer (but fixed for the function).
8. Write a function that expects at least n arguments, where n is a positive integer corresponding to the first argument.
9. Write a function that accepts an arbitrary number of arguments exactly one of which must be named. (An argument that “must be named” is one whose formal name *must* be given if the function is called.)
10. Write a function that, depending on the value of the 2nd argument, which should be a character scalar, computes either the minimum, the mean, or the maximum of its first argument (assumed to be a numeric vector). Let the mean be the default.
11. Write a function that takes a function `fun` as single argument and returns its arity (the “arity” of a function is the number of arguments that can be passed to it). Return `Inf` if there is no limit regarding the number of arguments passed to `fun`.
12. Write a function that accepts an arbitrary number of arguments and returns its call as

character scalar.

13. Write a function that returns itself.
14. Implement a function that returns a second function, which in the first call returns 1 and in each subsequent call the next higher integer (basically a counter function). Then implement the same function in a more flexible way, allowing the user to explicitly set the start and the increment values (but keeping the default of 1 in either case).
15. Implement a function that takes another function `fun` as argument, reverts the order of arguments of `fun` and returns the accordingly modified function.

Error handling

Creating and handling conditions that indicate unusual situations during program executions is essential for state-of-the-art programming, particularly in large projects. R offers mainly errors and warnings as predefined conditions (with `stop()` and `warning()` as the underlying functions), and a try-catch mechanism, implemented using `tryCatch()` for dealing with these conditions (and optionally also with user-defined ones) by passing specific handlers together with the expression to evaluate to this function. These handlers can simply be given as named function arguments, with the name indicating the condition for which the handler is provided. `stopifnot()` and `try()` are simpler but less customizable alternatives. The following example function attempts to convert an object `x` to a a numeric vector but returns `x` unchanged if a warning or even an error occurs:

```
try_numeric <- function(x, ...) { # example code #2
  tryCatch(expr = as.numeric(x), warning = function(w) x,
    error = function(e) x, ...)
}
```

Here `...` is used to optionally pass additional handlers. Because “warning” and “error” are classes that inherit from the abstract class “condition”, one could also state `condition = function(cond) x`, but this might conflict with other handlers to be passed. Note that a “finally” argument can be given to `tryCatch()` with code guaranteed to always be executed as its last operation. Similarly, `on.exit()` can be used within a function to guarantee code execution irrespective of whether or not an error occurs during the function call. (A typical example is the guaranteed resetting of global options that had to be modified by the function.)

Exercises

1. Write a function `unpercent()` that expects a numeric vector `x` as argument and divides it by 100. The function should raise an error with a meaningful message if `x` is not numeric and issue a warning with a meaningful message if any of the values in `x` are below 0 or above 100. Implement one version using `stop()` and one using `stopifnot()`. Which one would you prefer in a real project?
2. Write a function `must()` that takes an arbitrary expression as first argument and returns the results of this expression if neither errors nor warnings occur. If warnings occur, the function

should convert them to errors with the same message text. (This is actually the behaviour of a function from the **pkgutils** package used, e.g., by **opm**.)

3. Write a function `taste()` that takes an arbitrary expression as first argument and returns the results of this expression if no errors occur. If an error occurs, the function should return the error's message text as character scalar.
4. Write a function `relaxed()` that takes an arbitrary expression as first argument and returns the results of this expression if no errors occur. If a warning occurs, the function should turn the warning into a message using the warning's message text, and then return the result of evaluating the expression. (Hint: have a look at how `suppressWarnings()` is implemented.)

Attributes

In contrast to most other languages, R allows an arbitrary set of "attributes" to be added to arbitrary objects. The only real exception is `NULL`, which cannot have attributes; also note that the modification of certain attributes of certain objects can have destructive effects. Attributes add a lot of flexibility to the language, and it is no wonder that both object-oriented approaches in R, S3 and S4, have been implemented using attributes. Attributes are set and received individually using the `attr()<-` and `attr()` functions, respectively, and set and received at once using `attributes()<-` and `attributes()`, respectively. (There is also `mostattributes()<-`, which ensures the correct dimensions of objects that rely on them, such as matrices.) A lot of important functionality, such as names of vectors, row and column names of matrices and data frames, dimensions of matrices and arrays, and class names of objects with explicitly set classes, are implemented using attributes. For instance, the code `x <- 1:5; names(x) <- letters[x]` guarantees that we can access the vector elements with character keys, too, using e.g. `x["a"]`, because these keys are stored as "names" attribute.

Thus, frequently used attributes deserve their own getter and setter functions. Getter functions are just wrappers for `attr()`, whereas setter functions need the special semantics of R replacement functions. For instance, to set an attribute "author", the following convenience function could be defined:

```
`author<-` <- function(x, value) { # example code #3
  attr(x, "author") <- value
  x
}
```

Keep in mind that the replacement argument *must* be called "value", and that a replacement function *must* return the modified object.

Knowledge questions

1. Why has the name of the setter function shown above to be enclosed in backticks?
2. If you enter `y <- author(x) <- "George W. Bush"`, what is the value of `y`?
3. How can one delete (i) a single, selected attribute; (ii) all attributes at once?

Exercises

For some of the following exercises, the matrix object `m` is used, constructed as follows:

```
m <- matrix(1:10, ncol = 2) # example code #4

rownames(m) <- letters[1:5]

colnames(m) <- LETTERS[1:2]
```

1. Write a getter and a setter function for an attribute called “feature” (with arbitrary content).
2. Which attributes does `m` have, and what are they good for? Why is there no “class” attribute?
3. Convert `m` to a data-frame object `m2d` by entering `m2d <- as.data.frame(m)`. Study its attributes. Which are novel and what do they mean? Which other ones have changed or kept their meaning?
4. Can you convert `m` to a vector by removing something? If so, which attributes remained?
5. Reimplement code example #3 using `structure()`.
6. Metaprogramming: Write a function `set_getter_and_setter()` that takes the name of an attribute as first argument and creates getter and setter functions for it (like the ones you have defined in exercise #1) in the environment in which it is called.

Object-oriented programming with S3

Unfortunately, neither S nor R were designed to support object-oriented programming from the very beginning. As in many other languages which have later on been upgraded to this paradigm, a lot of aspects of object orientation are not as straightforward as they could be. I have compiled the following (probably incomplete) list of problems:

- Large parts of base R are written in a purely procedural style. This makes it less easy to predict (and learn) the behaviour of many important R utilities (particularly whether they use a formally defined method dispatch or do this internally). Likewise, classes are divided in implicit ones (treated in the chapter “R's basic types”) and explicit ones (all others), often making more checks necessary than in a more uniformly designed system.
- There are two competing implementations of object-orientation in R, S3 and S4. Again this results in the need for more checking, and also for more conversion functionality.
- Classes in S3 are informal; class names are not guaranteed to be associated with certain data types or combinations thereof. This might also result in the need for more checking.
- S4 is more formal and more elaborate but also results in more code overhead. Furthermore, the automated generation of documentation using Roxygen-like tools does not support S4.
- Object-oriented programming in R is function-centered, not class-centered. This is unusual for programmers experienced in other object-oriented programming languages. It is annoying particularly in S4 which requires the formal definition of both generic functions and classes. Moreover, this implementation via generic functions forces methods for the same generic function to contain the same formal arguments, at least those used in the definition of the generic function.

- In S3, dots in function names get a syntactical meaning. It is thus not directly possible to predict from a function name whether or not it is a method for a generic function.
- S3 heavily relies on `...` for passing arguments between methods because it is necessary for allowing methods to add own arguments which are not defined in the generic function. Function arguments with misspelled names will thus silently be ignored.
- S3 does not allow for formally defined multiple inheritance, virtual classes, mixins etc.

Having said this, I hasten to add that S3 actually turns out to be rather easy to use in practice. As long as the programmer keeps in mind that S3 classes are based on trust, and destructive actions are avoided, particularly the removal of parts of an object that are needed to ensure the appropriate behaviour of members of its class, serious problems are unlikely to occur. S3 implies little code overhead besides the usual one-liners for defining a generic function, and if one can cope with predefined objects such as those listed in Table 1 as the underlying objects (e.g., lists holding the needed components) to which just some special behaviour has to be added, S3 might be the preferable solution. In contrast, S4 is the better choice if

- more and stricter checking must be done during object construction;
- more overall (and particularly better defined) inheritance shall be used;
- objects are too complex to be easily modelled as lists or other basic objects;
- information has to be hidden more carefully;
- dispatch on multiple formal or on “missing” arguments is of interest;
- the problems related to the `...` argument shall be avoided.

But of course choice also depends on whether or not you have to operate in a predefined S3 or S4 coding environment. S4 is not further covered in this course (but see, e.g., **opm** for an example).

Implementing a generic “Hello world” function and its methods

The challenging thing about “hello world” (whose treatment was suggested by one of the course members, by the way) is that it is hard to make any sense of object-orientation in conjunction with it, but we will do our very best. First, let us define an S3 generic function:

```
hello_world <- function(x) { # example code #5

  UseMethod("hello_world")

}
```

This is all we need for getting started – `UseMethod()` will do the method dispatch for us and automatically pass its own arguments to the selected method. (For later on, note that because we do not define a `...` argument in the generic we cannot add further arguments to any of its methods.) The next step would be to define such methods, e.g. for the class “character”:

```
hello_world.character <- function(x) { # example code #6

  print(sprintf("Hello world, my name is '%s'!", x))

}
```

(Note the use of the C-style `sprintf()`, which often yields more compact code than `paste()`.) We can now enter, e.g., `hello_world(x = "Fred Firestone")`; then, `UseMethod("hello_world")` will search for a method whose name starts with "hello_world" followed by a dot and the name of the class, or one of the classes, of object `x` – "character" in our case. But, e.g., `hello_world(x = 5)` would fail because we have not yet defined a method for the class "numeric". A default method for `hello_world()` can be defined as follows:

```
hello_world.default <- function(x) { # example code #7
  print("Hello world!")
}
```

If `UseMethod()` cannot find a method for the class (or any of the classes) of the object passed to it, it searches for a method named "hello_world.default" and calls it. Only if such a default method is not found, an error results. One should now be able to enter `hello_world(5)` without receiving an error message.

It is now your task to define `hello_world()` methods for other predefined classes.

Exercises

1. Implement the `hello_world()` method for the "numeric" class as follows: Raise an error if the numeric vector `x` passed to it has zero length or contains negative numbers; otherwise print the friendly message `x` times.
2. Implement the `hello_world()` method for data frames as follows: Print the message "Hello world, I am a data frame with i rows and j columns!" but with "i" and "j" replaced by the correct numbers.
3. Implement the `hello_world()` method for factors by treating them like character vectors. (Hint: Do not repeat yourself!)

Implementing a "Hello world" class

Instead of defining a `hello_world()` generic function, we will now look at "Hello World" from the class perspective. We define only a single method for our class "hello.world", namely for the prominent function `print()`, which is already generic:

```
print.hello.world <- function(x, ...) { # example code #8
  print("Hello World!")
}
```

Note that we must use `x` and `...` as formal argument because these are the ones used by the `print()` generic. Now we only have to keep in mind that explicit classes are attributes, returned using `oldClass()` (the more frequently used `class()` returns implicit classes, too), and set using `class()<-`. S3 does not require specific constructor functions (that is why it is called an "informal" approach) – objects get members of a class if the name of the class is added to the

attribute named “class”. This knowledge allows us to write two helper functions for inserting a “hello.world” entry in the vector of classes of an object and returning a novel object:

```
insert_class <- function(x, klass) { # example code #9
  if (isS4(x))
    stop("this function is not intended for S4 objects")
  if (!identical(klass, make.names(klass))
    stop("'klass' contains invalid names")
  class(x) <- c(klass, oldClass(x))
  x
}
make_hello_world <- function(x) { # example code #10
  insert_class(x, "hello.world")
}
```

The second check in `insert_class()` is not absolutely necessary but avoids the generation of strange-looking class names.

The following exercises will first look at class-constructor functions that come with R, then examine whether our hello-world efforts will make our objects more friendly, and then switch to examining some generic functions and methods from the **yarp** package.

Exercises

1. Have a look at the last lines of the function bodies of `data.frame ()`, `factor()` and `table()` from **base** and `glm()` from **stats**. What do they have in common? Why?
2. Apply `make_hello_world()` to some R objects. How is their behaviour changed?
3. What happens if you replace `oldClass(x)` by `class(x)` in `insert_class()`?
4. What could happen if you replace `c(klass, oldClass(x))` by `c(oldClass(x), klass)` in `insert_class()`? Try this for a character vector and a data frame.
5. Implement `delete_class()`, which removes `klass` from `x`, and accordingly implement `unmake_hello_world()`.
6. Compare your `delete_class()` function to `unclass()`. Apply `unclass()` to a data frame. What do you get?
7. Rewrite the “arity” function (exercise #11 of “Writing functions in R”) as a generic function with a method for the class “function”, and rewrite the `unpercent()` function (exercise #1 of “Error handling”) as a method for the class numeric. Can you now simplify the body of this function?
8. Unzip the package **yarp**. Within its file “asqr.R” in the “R” subdirectory a generic function

`asqr()` has been defined. Determine for which classes methods of it have been defined and why precisely for these classes and in this way. Explain the use of the `...` operator in this context.

9. Examine in the same way `box_cox_fun()` and `box_cox()` in the file “box_cox.R”.
10. As an aside, call `box_cox_fun()` with a single `y` value. Where does the resulting function store this value? What does this have to do with exercise #14 of the “Writing functions in R” section?

S3 group generics

Group generics are probably the only slightly more complex, but also more powerful aspect of S3. Basically they allow one to define several methods at once by defining a virtual group method (which cannot be called directly). One of the group generics is `Summary`, which includes the functions `all()`, `any()`, `sum()`, `prod()`, `min()`, `max()` and `range()`. The following is a straightforward implementation for our “Hello World” class:

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

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

  NextMethod()

}
```

Beyond the fact that we can define group generics, this mainly demonstrates two issues. First, `UseMethod()` creates an environment for the method it selects with a number of special variables such as `.Generic`, which is a character scalar holding the name of the generic function that has been called. Second, `NextMethod()` is used to pass control to the eponymous function of the parent class (which is just the next entry in the “class” attribute, or an underlying implicit class). Like `UseMethod()` it automatically receives the arguments of the function that calls it.

Do not confuse the `Summary` group generic with the generic function `summary()`.

Exercises

1. Create a numeric vector and calculate its minimum and maximum. Then set its class to “hello_world” using `class()<-`, calculate the minimum and maximum again and watch the difference.
2. Create a corresponding “hello_world” method for the `Ops` group generics and test it.

Creating R packages

You should have received a copy of **yarp** (“Yet Another R Package”) and an independent copy of its PDF manual. Please do not install it into R but unpack it for studying its files and subdirectories. The structure of the package should be like this:

```
yarp/
```

```
yarp/DESCRIPTION

yarp/NAMESPACE

yarp/R/

yarp/R/asqr.R

yarp/R/box_cox.R

yarp/R/helpers.R

yarp/man/

yarp/man/asqr.Rd

yarp/man/box_cox.Rd

yarp/man/box_cox_fun.Rd

yarp/man/is_TF.Rd

yarp/man/simplify_conditionally.Rd
```

Apparently there are two files at the top level, “DESCRIPTION” and “NAMESPACE”, in addition to two subdirectories “R” and “man” containing only *.R and *.Rd files, respectively. The “DESCRIPTION” file is the major organizer for the package, formatted like a Debian Control File (DCF). The “NAMESPACE” file controls which functions, generic functions and methods are exported from the package (those not listed therein are kept internal). It uses a syntax that superficially looks like R code but is parsed in a much simpler way. The “R” subdirectory contains all R code; hence all files therein should use “.R” as file extension. The “man” subdirectory contains the documentation in Rd format (which is a specialized subset of LaTeX); hence all file names end in “.Rd”. All package documentation (PDF, HTML, and R-internal man pages) is inferred from such files.

Documenting R packages

The good news about R documentation is that once Rd files are available, they can be automatically converted to nicely formatted, interlinked PDF and HTML files. Moreover, example code can be placed in Rd files, which is automatically checked when running `R CMD check` on the package. Finally, Rd contains a large number of instructions not only for formatting but also for including specific information such as literature references.

The bad news is that Rd files have to be kept separate from the R files they document. This is particularly annoying for programmers, who want to have the documentation as close to the respective code as possible to minimize the risk of getting both out of sync. Fortunately, tools such as the **roxygen2** package allow Rd documentation to be automatically generated from specialized comments in R code files. **roxygen2** uses tags such as `@param`, which are converted to according Rd instructions (`\arguments{}` and `\item{}` in that case). A package just has to be “roxygenized” before running `R CMD check`, which, as a by-product, generates a PDF manual. “Roxygenizing” can be conducted non-interactively by running an Rscript-based script on the operation systems's command line.

Exercises

1. Clarify what the entries in DESCRIPTION are good for. Compare this file's content with the first page of the PDF manual.
2. Which instructions are contained in NAMESPACE and what do they mean?
3. Compare the documentation of the `asqr()` function in the PDF manual with the content of the file “asqr.Rd” and clarify which instruction within this Rd file induces which part of the PDF documentation.
4. Compare the content of the file “asqr.Rd” with the Roxygen-style documentation of the `asqr()` function within the file “asqr.R” and clarify which instruction within this R file induces which part of the Rd file documentation. (Hint: regarding `@family`, have a look at the links within the PDF documentation of `asqr()`.)
5. Compare the instructions in NAMESPACE for the `asqr()` function and its methods with its Roxygen-style documentation within the file “asqr.R” and clarify which Roxygen tag causes which NAMESPACE entry to be written.
6. Have a look at the functions within the file “helpers.R”. Apparently they do not occur in NAMESPACE. What does this imply regarding their user-visibility once the package is loaded? Which Roxygen entry causes these functions to be omitted from NAMESPACE?
7. Delete the “yarp/man” subdirectory. Then use the “docu.R” of **pkgutils** script to generate the documentation of **yarp** again. Study the source code of “docu.R” for how scripting using Rscript works.
8. Also on the command-line, run `R CMD check` with the **yarp** package. This should generate a directory “yarp.Rcheck”. Study its content, particularly “00check.log” and “yarp-Ex.Rout”.
9. Unpack the **opm** package. Have a look at its top-level files. What are all the files and subdirectories of **opm** good for that are lacking from **yarp**?

References

In the author's view, the strength of the following list lies in what it does *not* contain. Most introductions into R focus on how to do this and that statistical stuff with it. As a logical consequence, they will distract a programmer from the necessary facts. For obvious reasons, introductions into R are likely to fall for this “statistics trap”, with the consequence that many long-time interactive users of R (even if they are experienced programmers in other languages) have actually not much clue about what “programming in R” could mean. The following list is thus short, and there may be many other valuable resources out there, but the author's experience only allows these ones to be wholeheartedly recommended:

- S3 programming: R. Gentleman, R Programming for Bioinformatics. Chapman & Hall 2009, chapter 3.
- Creating packages: http://cran.r-project.org/doc/manuals/R-exts.pdf (useful hints regarding S3 programming are in chapter 7)
- R for programmers: http://heather.cs.ucdavis.edu/~**matloff**/R/RProg.pdf

Appendix 1: Comments on a selection of useful functions

This section is mainly intended as a list of names of useful functions. We will not provide much details about heir functioning but mainly comment on whether or not they are generic.

Environment

See `Sys.getenv()`, `Sys.setenv()` and the help for the topic “environment variables” for use and modification of such variables. Regarding file handling, see the topic “files” as well as `getwd()` `setwd()`, `list.files()`, `file.exists()` and finally `file_test()` from the **utils** package.

IO

See `read.table()` and its companions. Your own input functions can be based on `scan()` and `readLines()`. For output `write.table()` is available as well as the lesser specialized `write()` and `cat()`. For within sessions consider `sink()` and `capture.output()`. R objects themselves can be output/input with `save()`/`load()` and `dump()`/`source()` using binary or textual format (R code), respectively. None of this is written in an object-oriented style.

Common vector operations

For generating vectors, in addition to the constructor functions listed in Table 1, `c()` is frequently used. It is generic, but only for S4 methods. For sorting, there are `sort.int()` (for all kinds of vectors) and its object-oriented companion `sort()`. `rank()` is also available but not generic. `order()` and `sort.list()` (which is not a sorting function for lists!) are useful if an object should be sorted according to the sorting order of one to several other objects. `unique()`, `rev()`, `duplicated()` and `anyDuplicated()` are generic functions (and thus not restricted to vectors). For generating regular sequences, there are the internally generic `seq.int()` and its “normal” object-oriented companion `seq()`. `rep()` is also internally generic and thus has restrictions regarding the definitions of new methods. Method definition for such functions is via S4 and cannot be done for the implicit classes, which would appear as “sealed”. See the help for the topic “InternalMethods” for further details. The common `names()` and `` `names<-`() `` functions work in the same way. A “normal” generic function with, thus, fewer restrictions is `labels()`. `tapply()` is not generic. Finally, note that `seq_along()` and `seq_len()` are often useful in loops and preferable to constructions using `length()` and the `` `:` `` operator.

Sets

A simple approach, treating vectors as sets, is available, with the predefined functions `union()`, `intersect()`, `setdiff()` and `setequal()`. These are all weakly typed in the sense that all of them first call `as.vector()`.

Logical vectors

For the `Summary` group generic, see above. Remaining information is available in the help for the topic “Logic”. You will also frequently need `which()`, but note that `which.min()` and `which.max()` are available for use with numeric vectors.

Character vectors

Predefined functions for processing character vectors are not written in an object-oriented style, and most of them are weakly typed in the sense that they start with converting the passed objects using `as.character()`. `paste()` and `sprintf()` are useful for creating character vectors. `grep()`, `grepl()`, `regexpr()`, `gregexpr()` and `regexec()` can be used to match character vectors with regular expressions, and `agrep()` can do this error-tolerantly. `match()`, `charmatch()` and `pmatch()` match using full or partial fixed strings, but note that the infix operator `%in%` is often more useful in conditions. Among those functions, `match()` and `%in%` do not convert to character mode unless necessary and are thus general vector operators. `nchar()` and `nzchar()` are used to get string lengths or test for non-zero lengths, respectively. For modifying character vectors, `sub()`, `gsub()`, `substr()`, `` `substr<-`() ``, `strsplit()`, `toupper()` and `tolower()` are available, with the expected meaning.

Matrices and data frames

`dim()`, `dimnames()` and their replacement functions are internally generic and thus have restrictions regarding the writing of new methods (see above). The same holds for the bracket and dollar-sign operators, but `subset()` is a “normal” generic function. `nrow()` and `ncol()` just rely on `dim()`, but there are also `NROW()` and `NCOL()`, which can sensibly be applied to vectors, too. `t()` is generic and, e.g., also available for data frames. `rownames()` and `colnames()` as well as `row()` and `col()` are not generic but can be applied to any matrix-like (two-dimensional) object. `rbind()` and `cbind()` are also generic but not via `UseMethod()`. Their behaviour is thus also special. `rowSums()`, `colSums()`, `rowMeans()` and `colMeans()` are all weakly typed as they attempt to convert the passed objects using `as.matrix()`. For use in programming, these four functions have faster but less safe companions whose names start with a dot. As the next step, one frequently applies `sweep()` (which is not generic). `rowsum()` as well as the more general `aggregate()` and `by()` are generic. `with()`, which is a safer alternative to `attach()`, is also generic.

Functional programming

For higher-order function-programming methods, see `Reduce()` and its companions `Filter()`, `Find()`, `Map()` and `Position()`. Regarding the apply-family of functions, note particularly that for programming `vapply()` is usually safer than `sapply()` because, e.g., the latter might return the wrong type if the input list is empty.

Appendix 2: R Programming Style Guide

The goal of the R Programming Style Guide is to make our R code easier to read, share, and verify. (Assuming that no one within DMSZ has produced one before, I have tentatively called it DSMZ R Style Guide Version 1.) The rules below were compiled by me after consulting Google's R style guide (http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html) and the CRAN manuals on writing R extensions (http://cran.r-project.org/doc/manuals/R-exts.html) and Rd files (http://developer.r-project.org/Rds.html). They had to be modified, however, e.g., for resolving conflicts. Google's R style guide did neither seem to consider nor seem to be compatible with writing R packages, and thus the entire "General Layout and Ordering" section had to be deleted.

Notation and Naming

File Names. File names should end in ".R" and, of course, be meaningful.

```
# GOOD:

predict_ad_revenue.R

# BAD:

foo.R
```

I personally use a scheme in which all non-internal functions of a package are placed that has the same name than the family to which the function is assigned using **roxygen2**'s `@family` tag. For instance, the file "plotting.R" would contain functions with the entry `@family plotting-functions`.

Identifiers. Identifiers should be named according to the following conventions. Variable names should have all lower case letters and words separated with dots (`.`); function names should have all lower case letters and words separated with underscores (`_`); dots are only used in S3 generic function definitions; constants are in all upper case words separated with underscores (`_`). Regarding function and constant names, this deliberately deviates from Google's R style guide. Its suggestion to use camel case in function names does not seem idiomatic in most R packages; also, underscores are more readable than camel case. I have observed them in many recent R packages.

```
# GOOD:

variable.name, avg.clicks

# BAD:

avg_Clicks, avgClicks

# GOOD:

function_name, calculate_avg_clicks

# BAD:

CalculateAvgClicks, calculateAvgClicks

# GOOD:
```

`CONSTANT_NAME`

Function names should be made verbs. An exceptions are constructor functions for classed objects, in which case the function name (constructor) and the class should match (e.g., `opms()` from the **opm** package).

Syntax

Line length. The maximum line length is 80 characters. (The Rd style guide at http://developer.r-project.org/Rds.html even recommends at most 65 characters for example R code.)

Indentation. When indenting your code, use two spaces. Experience with other languages shows that two are entirely sufficient. Never use tabs or mix tabs and spaces. Continuation lines are also indented with two spaces only, use four only if the line ends with '{' because in that case the next line has to be indented with two spaces. This deliberately deviates from Google's R style guide for avoiding unnecessary work just for beautifying the code. (The current indentation default of Rstudio, http://www.rstudio.org/, is also suboptimal because it wastes much space and does not result in files with a consistent layout anyway.)

Spacing. Place spaces around all binary operators (=, +, -, <-, etc.). This greatly increases readability. Do not place a space before a comma, but always place one after a comma (just as you would when writing a sentence in a natural language!).

```
# GOOD:

tab.prior <- table(df[df$days.from.opt < 0, "campaignid"])

total <- sum(x[, 1L])

total <- sum(x[1L, ])

# BAD:

# Needs spaces around '<'

tab.prior <- table(df[df$days.from.opt<0, "campaignid"])

# Needs a space after the comma

tab.prior <- table(df[df$days.from.opt < 0,"campaignid"])

# Needs a space before <-

tab.prior<- table(df[df$days.from.opt < 0, "campaignid"])

# Needs spaces around <-

tab.prior<-table(df[df$days.from.opt < 0, "campaignid"])

# Needs a space after the comma

total <- sum(x[,1L])

# Needs a space after the comma, not before

total <- sum(x[ ,1L])
```

Place a space before a left parenthesis, except in a function call. This help to distinguish flow control from function calls.

```
# GOOD:

if (debug)

# BAD:

if(debug)
```

Do not place spaces around code in parentheses or square brackets. As an exception, always place a space after a comma (the reason was given above).

```
# GOOD:

if (debug)

x[1L, ]

# BAD:

if ( debug ) # No spaces around debug

x[1L,] # Needs a space after the comma
```

Curly braces. An opening curly brace should never go on its own line; a closing curly brace should always go on its own line. You may omit curly braces when a block consists of a single statement; however, you must consistently either use or not use curly braces for single-statement blocks.

```
if (is.null(ylim)) {

  ylim <- c(0, 0.06)

}

# xor (but not both)

if (is.null(ylim))

  ylim <- c(0, 0.06)

Always begin the body of a block on a new line.

# BAD:

if (is.null(ylim)) ylim <- c(0, 0.06)

if (is.null(ylim)) {ylim <- c(0, 0.06)}
```

Assignment. Use <-, not =, for assignment. One reason is that = has a distinct meaning when using it for assigning named function arguments, and thus being consequent here avoids confusion.

```
# GOOD:

x <- 5

# BAD:

x = 5
```

Semicolons. Do neither terminate your lines with semicolons nor use semicolons to put more than one command on the same line.

Organization

Commenting Guidelines. Comment your code in English. Avoid special characters such as German umlauts. Entire commented lines should begin with '#' and one space. Short comments can be placed after code preceded by one space, '#', and then one space. roxygen2 comment lines have to start with '#', a single quote, and one space.

```
# Create histogram of frequency of campaigns by percent budget spent.

hist(df$pct.spent,

  breaks = "scott",  # method for choosing number of buckets

  main = "Histogram: fraction budget spent by campaignid",

  xlab = "Fraction of budget spent",

  ylab = "Frequency (count of campaignids)")
```

Function definitions and calls. Function definitions should first list arguments without default values, followed by those with default values. In both function definitions and function calls, multiple arguments per line are allowed; line breaks are only allowed between assignments.

```
# GOOD:

predict_ctr <- function(query, property, num.days,

    show.plot = TRUE) {

# BAD:

predict_ctr <- function(query, property, num.days, show.plot =

    TRUE) {
```

Note that we here indent with four spaces because this is a special kind of continuation line (see above).

Ideally, unit tests should serve as sample function calls (for shared library routines). If code is organized in packages, examples in the manual should (also) serve this purpose.

Function Documentation. Functions are either part of a package or part of a file that can be read using `source()`. For writing packages, we recommend documentation in **roxygen2**-style, even though **roxygen2**'s support for S4 methods is poor (see **pkgutils** for remedies). For sourced functions, you might use Google's R style guidelines.

TODO Style. Use a consistent style for TODOs throughout your code.