summary

sara

10/17/2017

1. Functions

Function components

The three components of a function are its body, arguments, and environment.

1. body(), the code inside the function.
2. formals(), the list of arguments which controls how you can call the function.
3. environment(), the map of the location of the function's variables.

When you print a function in R, it shows you these three important components. If the environment isn't displayed, it means that the function was created in the global environment.

```
f <- function(x) x^2
f
```

```
## function(x) x^2
```

```
body(f)
```

```
## x^2
```

```
formals(f)
```

```
## $x
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

The assignment forms of body(), formals(), and environment() can also be used to modify functions.

Like all objects in R, functions can also possess any number of additional attributes()

You can also add attributes to a function. For example, you can set the class() and add a custom print() method.

Primitive functions

There is one exception to the rule that functions have three components.

Primitive functions, like sum(), call C code directly with .Primitive() and contain no R code.

Therefore their formals(), body(), and environment() are all NULL

```
#sum
#>function (..., na.rm = FALSE)  .Primitive("sum")
```

Lexical scoping

Scoping is the set of rules that govern how R looks up the value of a symbol.

Lexical scoping looks up symbol values based on how functions were nested when they were created, not how they are nested when they are called.

With lexical scoping, you don't need to know how the function is called to figure out where the value of a variable will be looked up. You just need to look at the function's definition.

There are four basic principles behind R's implementation of lexical scoping:

- name masking
- functions vs. variables
- a fresh start
- dynamic lookup

Name masking

If a name isn't defined inside a function, R will look one level up.

The same rules apply if a function is defined inside another function: look inside the current function, then where that function was defined, and so on, all the way up to the global environment, and then on to other loaded packages.

The same rules apply to closures, functions created by other functions:

```
j <- function(x) {
y <- "kalas"
function() {
paste0(x, y)
}
}
k <- j("Barn")
k()
```

```
## [1] "Barnkalas"
```

k preserves the environment in which it was defined and because the environment includes the value of y.

Functions vs. Variables

The same principles apply regardless of the type of associated value — finding functions works exactly the same way as finding variables

For functions, there is one small tweak to the rule. If you are using a name in a context where it's obvious that you want a function (e.g., f(3)), R will ignore objects that are not functions while it is searching.

In the following example n takes on a different value depending on whether R is looking for a function or a variable.

```
n<- function(x) x / 2
o <- function() {
n <- 10
n(n)
}
o()
```

```
## [1] 5
```

However, using the same name for functions and other objects will make for confusing code, and is generally best avoided.

A fresh start

Every time a function is called, a new environment is created to host execution. A function has no way to tell what happened the last time it was run; each invocation is completely independent.

Dynamic lookup

R looks for values when the function is run, not when it's created. This means that the output of a function can be different depending on objects outside its environment

You generally want to avoid this behaviour because it means the function is no longer self-contained. This is a common error

One way to detect this problem is the findGlobals() function from codetools. This function lists all the external dependencies of a function:

```
f <- function() x + 4
codetools::findGlobals(f)
```

```
## [1] "+" "x"
```

Every operation is a function call

Everything that exists is an object. Everything that happens is a function call.

Function arguments

Calling a function given a list of arguments

List of function arguments to send to mean()

```
args <- list(x=1:10, na.rm = TRUE)
```

Send list to mean with do.call:

```
do.call(what = "mean", args = args)
```

```
## [1] 5.5
```

Lazy evaluation

By default, R function arguments are lazy — they're only evaluated if they're actually used

```
f <- function(x) {
10
}
f(stop("This is an error!"))
```

```
## [1] 10
```

If you want to ensure that an argument is evaluated you can use force()

```
f <- function(x) {
force(x)
10
}
#f(stop("This is an error!"))
```

Special calls

R supports two additional syntaxes for calling special types of functions: infix and replacement functions.

Infix

The function name comes in between its arguments

```
# 1+1
```

Replacement functions

```
`second<-` <- function(x, value) {
x[2] <- value
x
}

x <- 1:10
second(x) <- 5L
x
```

```
##  [1]  1  5  3  4  5  6  7  8  9 10
```

It's often useful to combine replacement and subsetting

Return values

The last expression evaluated in a function becomes the return value, the result of invoking the function.

```
hej <- function(){
  "Banan"
  "Gurka"
}
hej()
```

```
## [1] "Gurka"
```

R protects you from one type of side effect: most R objects have copy-on-modify semantics. So modifying a function argument does not change the original value

(There are two important exceptions to the copy-on-modify rule: environments and reference classes. These can be modified in place, so extra care is needed when working with them.)

Functions can return invisible values, which are not printed out by default when you call the function. You can force an invisible value to be displayed by wrapping it in parentheses, (f())

On exit

As well as returning a value, functions can set up other triggers to occur when the function is finished using on.exit(). This is often used as a way to guarantee that changes to the global state are restored when the function exits. The code in on.exit() is run regardless of how the function exits, whether with an explicit (early) return, an error, or simply reaching the end of the function body.

2. Environments

The environment is the data structure that powers scoping. This chapter dives deep into environments, describing their structure in depth, and using them to improve your understanding of the four scoping rules described in lexical scoping

Environments can also be useful data structures in their own right because they have reference semantics. When you modify a binding in an environment, the environment is not copied; it's modified in place.

The parent of the global environment is the last package that you loaded. The only environment that doesn't have a parent is the empty environment.

The enclosing environment of a function is the environment where it was created. It determines where a function looks for variables.

To determine the environment from which a function was called - use parent.frame

Environment basics

The job of an environment is to associate, or bind, a set of names to a set of values. You can think of an environment as a bag of names

Each name points to an object stored elsewhere in memory

```
sara <- new.env()
sara$a <- "Kanelbullar"
sara$b <- 1:4
sara$c <- 2.9
sara$d <- TRUE
```

The objects don't live in the environment so multiple names can point to the same object:

```
sara$d <- sara$b
```

Confusingly they can also point to different objects that have the same value:

```
sara$d <- 1:4
```

If an object has no names pointing to it, it gets automatically deleted by the garbage collector.

Every environment has a parent, another environment.

The parent is used to implement lexical scoping: if a name is not found in an environment, then R will look in its parent (and so on). Only one environment doesn't have a parent: the empty environment.

Ways that an environment is different to a list:

- every object in an environment must have a unique name
- order doesn't matter
- environments have parents
- environments have reference semantics.

An environment is made up of two components, the frame, which contains the nameobject bindings (and behaves much like a named list), and the parent environment.

There are four special environments:

- The globalenv(), or global environment, is the interactive workspace. This is the environment in which you normally work. The parent of the global environment is the last package that you attached with library() or require().
- The baseenv(), or base environment, is the environment of the base package. Its parent is the empty environment.
- The emptyenv(), or empty environment, is the ultimate ancestor of all environments, and the only environment without a parent.
- The environment() is the current environment.

search() lists all parents of the global environment.

To create an environment manually, use new.env(). You can list the bindings in the environment's frame with ls() and see its parent with parent.env().

```
parent.env(sara)
```

```
## <environment: R_GlobalEnv>
```

```
ls(sara)
```

```
## [1] "a" "b" "c" "d"
```

Another useful way to view an environment is ls.str(). It is more useful than str() because it shows each object in the environment.

```
str(sara)
```

```
## <environment: 0x7fc7fe8f9b88>
```

```
ls.str(sara)
```

```
## a :  chr "Kanelbullar"
## b :  int [1:4] 1 2 3 4
## c :  num 2.9
## d :  int [1:4] 1 2 3 4
```

Deleting objects from environments works a little differently from lists. With a list you can remove an entry by setting it to NULL. In environments, that will create a new binding to NULL. Instead, use rm() to remove the binding.

```
sara$d <- NULL
sara$d #NULL
```

```
## NULL
```

```
rm("d", envir = sara)
ls(sara)
```

```
## [1] "a" "b" "c"
```

You can determine if a binding exists in an environment with exists().

Function environments

Most environments are not created by you with new.env() but are created as a consequence of using functions.

Four types of environments associated with a function:

- enclosing
- binding
- execution
- calling

The enclosing environment is the environment where the function was created.

You can determine the enclosing environment of a function by calling environment()

Function inside a function

When you create a function inside another function, the enclosing environment of the child function is the execution environment of the parent, and the execution environment is no longer ephemeral.

```
power <- function(x){
  function(y) y^x
}

square <- power(2)
square(1:3)
```

```
## [1] 1 4 9
```

```
cube <- power(3)
cube(1:3)
```

```
## [1]  1  8 27
```

Binding names to values

Assignment is the act of binding (or rebinding) a name to a value in an environment

You've probably used regular assignment in R thousands of times. Regular assignment creates a binding between a name and an object in the current environment.

The regular assignment arrow, <-, always creates a variable in the current environment.

The deep assignment arrow, <<-, never creates a variable in the current environment, but instead modifies an existing variable found by walking up the parent environments.

You can also do deep binding with assign(): name <<- value is equivalent to assign("name", value, inherits = TRUE).

```
x <- 0
f <- function(){
  x <<- x+1
}
f() #x ökar varje gång funktionen körs
x
```

```
## [1] 1
```

If <<- doesn't find an existing variable, it will create one in the global environment. This is usually undesirable, because global variables introduce non-obvious dependencies between functions.

<<- is most often used in conjunction with a closure

Summa kardemumma:

<- always creates a binding in the current environment

<<- rebinds an existing name in a parent of the current environment.

3. Functional programming

You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

Three building blocks of functional programming: anonymous functions, closures (functions written by functions), and lists of functions

lapply() is called a functional, because it takes a function as an argument.

```
set.seed(1014)
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]

fix_missing <- function(x) {
x[x == -99] <- NA
x
}
df[] <- lapply(df, fix_missing)
```

Anonymous functions

If you choose not to give the function a name, you get an anonymous function.

```
(function(x) x^2)(10)
```

```
## [1] 100
```

Like all functions in R, anonymous functions have formals(), a body(), and a parent environment()

Closures

Closures = functions written by functions

Closures get their name because they enclose the environment of the parent function and can access all its variables.

This is useful because it allows us to have two levels of parameters: a parent level that controls operation and a child level that does the work.

Lists of functions

In R, functions can be stored in lists. This makes it easier to work with groups of related functions, in the same way a data frame makes it easier to work with groups of related vectors.

4. OO field guide

R has three object oriented systems (plus the base types)

Central to any object-oriented system are the concepts of class and method.

A class defines the behaviour of objects by describing their attributes and their relationship to other classes.

The class is also used when selecting methods, functions that behave differently depending on the class of their input.

Classes are usually organised in a hierarchy: if a method does not exist for a child, then the parent's method is used instead; the child inherits behaviour from the parent.

R's three OO systems:

- S3
- S4
- RC

1. How do you tell what OO system (base, S3, S4, or RC) an object is associated with?

To determine the OO system of an object, you use a process of elimination. If !is.object(x), it's a base object. If !isS4(x), it's S3. If !is(x, "refClass"), it's S4; otherwise it's RC.

2. How do you determine the base type (like integer or list) of an object?

Use typeof() to determine the base class of an object.

3. What is a generic function?

A generic function calls specific methods depending on the class of it inputs. In S3 and S4 object systems, methods belong to generic functions, not classes like in other programming languages.

4. What are the main differences between S3 and S4? What are the main differences between S4 & RC?

S4 is more formal than S3, and supports multiple inheritance and multiple dispatch.

RC objects have reference semantics, and methods belong to classes, not functions.

S3

S3 is R's first and simplest OO system. In S3, methods belong to functions, called generic functions. S3 methods do not belong to objects or classes.

Defining classes and creating objects

To make an object an instance of a class, you just take an existing base object and set the class attribute.

```
finlista <- list()
class(finlista) <- "finlista"
class(finlista)
```

```
## [1] "finlista"
```

```
klassigt <- foo <- structure(list(), class = "klassigt")
class(klassigt)
```

```
## [1] "klassigt"
```

You can determine the class of any object using class(x), and see if an object inherits from a specific class using inherits(x, "classname").

Creating new methods and generics

To add a new generic, create a function that calls UseMethod(). UseMethod() takes two arguments: the name of the generic function, and the argument to use for method dispatch.

```
print.klassigt <- function(x){
  paste0("This is my class, ", class(klassigt))
}
print(klassigt)
```

```
## [1] "This is my class, klassigt"
```

S4

Methods still belong to functions, not classes, but:

- Classes have formal definitions which describe their fields and inheritance structures (parent classes).
- Method dispatch can be based on multiple arguments to a generic function, not just one.
- There is a special operator, @, for extracting slots (aka fields) from an S4 object.

Defining classes and creating objects

In S3, you can turn any object into an object of a particular class just by setting the class attribute.

S4 is much stricter: you must define the representation of a class with setClass(), and create a new object with new()

```
setClass("student",
         slots = list(name = "character",
                      age = "numeric"))
pelle <- new("student", name="pelle", age=26)

setClass("teacher",
         slots = list(teach = "character"),
         contains = "student")
kalle <- new("teacher", name="kalle", age=80, teach="pelle")

kalle@age
```

```
## [1] 80
```

```
slot(kalle, "age")
```

```
## [1] 80
```

Creating new methods and generics

setGeneric() creates a new generic or converts an existing function into a generic.

setMethod() takes the name of the generic, the classes the method should be associated with, and a function that implements the method

RC

RC methods belong to objects, not functions

RC objects are mutable: the usual R copy-on-modify semantics do not apply

Defining classes and creating objects

Use setRefClass()

```
boss <- setRefClass("boss",
            fields = list(name = "character",
            age = "numeric"))
a <- boss$new(name="zlatan", age=65)

a$age <- 85
```

Note that RC objects are mutable, i.e., they have reference semantics, and are not copied-on-modify

```
b <- a
a$age <- 34
b$age #changes
```

```
## [1] 34
```

For this reason, RC objects come with a copy() method that allow you to make a copy of the object:

```
c <- a$copy()
c$name
```

```
## [1] "zlatan"
```

```
a$name <- "hej"
c$name
```

```
## [1] "zlatan"
```

RC methods are associated with a class and can modify its fields in place.

```
money <- setRefClass("money",
      fields = list(cash = "numeric"),
      methods = list(
        withdraw = function(x){
          cash <<- cash - x
        },
        deposit = function(x){
          cash <<- cash + x
        }
      ))
```

You call an RC method in the same way as you access a field

```
a <- money$new(cash = 500)
a$cash
```

```
## [1] 500
```

```
a$withdraw(499)
a$cash
```

```
## [1] 1
```

contains - the parent RC class to inherit behaviour from

```
nomoney <- setRefClass("nomoney",
contains = "money",
methods = list(
withdraw = function(x) {
if (cash < x) stop("Not enough money")
cash <<- cash - x
}
)
)

mymoney <- nomoney$new(cash=40)
mymoney$deposit(1)
mymoney$cash
```

```
## [1] 41
```

```
mymoney$withdraw(2) #works
#mymoney$withdraw(1000) #stops
```

```
library(ggplot2)
gen_shape<- function()
{
  t <- seq(from = 1, to = 2 * pi , by = 0.1)
  x_v <- 16 * (sin(t)^3)
  y_v <- 13 * cos(t) -5 * cos(2*t) - 2* cos(3* t) - cos(4*t)
  return (data.frame(t=t, x=x_v, y=y_v))
}


df <- gen_shape()

ggplot(df , aes(x= x , y = y), col= "red") + geom_polygon()
```


10
0
−10
y
−10
0
10
x