Functions and OOP in R

Kylie Ariel Bemis

10/31/2017

Functions and OOP in R

To understand computations in R, two slogans are helpful:

- *Everything that exists is an object*
- *Everything that happens is a function call*

— John Chambers, creator of S

"Everything that exists is an object"

Consider these simple lines of code:

```
x <- 2
y <- 3
x + y
```

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

What actually happens when you run `x + y`?

"Everything that exists is an object"

```
sexp <- quote(x + y)
sexp
```

```
## x + y
```

What type of object is `sexp`?

```
typeof(sexp)
```

```
## [1] "language"
```

`typeof` returns an object's **base type**.

"Everything that exists is an object"

R code itself is a language object that can be manipulated, changed, and evaluated.

```
sexp
```

```
## x + y
```

```
eval(sexp)
```

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

```
sexp[[1]] <- quote(`*`)
sexp
```

```
## x * y
```

```
eval(sexp)
```

```
## [1] 6
```

"Everything that exists is an object"

```
sexp
```

```
## x * y
```

```
as.list(sexp)
```

```
## [[1]]
## `*`
##
## [[2]]
## x
##
## [[3]]
## y
```

Why does the * come first in the object? (Hint: see next slide.)

"Everything that happens is a function call"

Reconsider:

```
x <- 2
y <- 3
x + y
```

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

This is the same as doing:

```
`<-`(x, 2)
`<-`(y, 3)
`+`(x, y)
```

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

“Everything that happens is a function call”

Why does this work?

```
`<-`(x, 2)
`<-`(y, 3)
`+`(x, y)
```

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

In R, addition is just a function for which we commonly use the **infix** notation, but using its **prefix** notation works just as well, and is how functions are internally stored in R.

(This is why we accessed `*` using `sexp[[1]]` instead of `sexp[[2]]`.)

Functions in R

Functions are first-class citizens in R. They are objects that can be passed around and manipulated like any other object.

Functions in R have three key characteristics:

- `body` – the code inside the function
- `formals` – a list of arguments used to call the function
- `environment` – where to find the function's variables

You provide `body` and `formals` when defining a function. The `environment` is defined automatically by **where you are** when you define it.

Functions in R (cont'd)

```
add <- function(x, y) x + y
body(add)
```

```
## x + y
```

```
formals(add)
```

```
## $x
##
##
## $y
```

```
environment(add)
```

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

Exception: primitive functions

Some low-level "primitive" functions defined by the core R team are exceptions to this, and call C code immediately upon being called. + is actually a primitive function.

```
`+`
```

```
## function (e1, e2)  .Primitive("+")
```

Primitive functions only exist in the `base` R package and can only be created by the core R team, so we won't discuss them any further beyond acknowledging their existence.

Defining a function in R

Functions in R are defined by the `function` function:

- The arguments you provide to `function` become the formal arguments of your function
- An expression follows that becomes the body of the function
- Your current environment becomes the environment for the function

```
add <- function(x, y) x + y
add
```

```
## function(x, y) x + y
```

Note that while you can explicitly return values with the `return` function, most R functions simply return the value of the last evaluated expression in the body. In our add function above, that is simply `x + y`.

Anonymous functions

Note that we don't actually have to assign the function to a variable to use it.

```
(function(x, y) x + y)
```

```
## function(x, y) x + y
```

```
(function(x, y) x + y)(1, 2)
```

```
## [1] 3
```

This is called an **anonymous function**. Anonymous functions are useful when using functions like `lapply`, `sapply`, and `purrr::map`.

Note that `purrr::map`'s allowing us to use a formula interface in place of a function is simply syntactic sugar for specifying an anonymous function more briefly.

Lexical scoping

How does a function find values for the variables in its `body`?

```
add_1 <- function(x) x + 1
add_1
```

```
## function(x) x + 1
```

```
add_1(1)
```

```
## [1] 2
```

```
add_y <- function(x) x + y
add_y
```

```
## function(x) x + y
```

It is clear what `add_1` does. But what will `add_y` do to find `y`?

Lexical scoping

Functions capture the environment in which they were created, and have access to all variables in the environment.

Because we created `add_y` in the global environment, that means it has access to all variables in the global environment.

We simply need to define a `y` variable in the global environment.

```
add_y
```

```
## function(x) x + y
```

```
environment(add_y)
```

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

```
y <- 2
add_y(1)
```

```
## [1] 3
```

Why would we want to do something like this?

Functionals

Suppose we wish to create a function that allows a user to add some number `val` to any number, but we don't know what `val` will be. We can simply create that function once we know what `val` is!

A function that returns a function like this (or takes a function as an argument) is called a **functional**. Functionals are common in R, most notably in functions like `lapply`, `sapply`, and `purrr::map`.

```
add_val <- function(val) {
  function(x) x + val
}
add_10 <- add_val(10)
add_10(1)
```

```
## [1] 11
```

What happened here?

Lexical scoping (cont'd)

When a function is called in R, the following happens:

- A new, temporary environment is created
- Any formal arguments of the function are assigned to the temporary environment
- The temporary environment's *parent environment* (or "enclosing" environment) is the *function's environment*
- The function is evaluated in this temporary environment
- When a variable name is encountered, R searches the current (temporary) environment, then its parent environment (the function's environment), then its parent's parent environment, and so on, until the variable is found

Lexical scoping and closures

```
add_val <- function(val) {
  function(x) x + val
}
add_10 <- add_val(10)
add_10
```

```
## function(x) x + val
## <environment: 0x7fb2db4a12a8>
```

When we evaluate `add_val`, it creates a temporary environment and assigns `val` into it. It then returns a new function whose environment *is* the "temporary" environment created by evaluating `add_val`, which is where `val` can be found. Now our new function `add_10` always has access to `val` (which in our example is 10).

When a function is stored together with its environment like this, it's called a **closure**.

Functions (cont'd)

There is a lot more that can be done with functions in R, but we don't need most of those things. See the **Advanced R** chapter on functions for more information on:

- Default arguments and missing arguments
- Lazy evaluation
- Special calls
  - Infix functions
  - Replacement functions

and more.

For now, we already know enough about functions to be able to talk about object-oriented programming in R.

Introduction to Object-Oriented Programming in R

Object-oriented programming (OOP) is a way of organizing code around commonly re-used data "classes" and "methods".

A **class** is a blueprint for a way of organizing data.

- E.g., you might write a class for a special type of data frame called a `tibble`.

An **object** is a particular instance of a class.

- E.g., the `flights` and `diamonds` datasets are particular instances of `tibbles`.

Using **inheritance** allows subclasses to specialize superclasses.

- E.g., a `tibble` inherits most of its behavior from `data.frame`.

A **method** is a function associated with behavior specialized to a particular class. In R this is done using **generic functions**.

- E.g., `filter` is a generic function. It works differently for `tbl_df` (a `tibble` stored in-memory) and `tbl_dbi` (a `tibble` stored in an on-disk database).

A simple example

Consider a pet simulator game. It may consist of the following elements:

- An `Animal` class with child classes `Cat` and `Dog`.
- A generic function called `speak`.
- A `speak` method for both the `Cat` and `Dog` classes
- An object named `Mittens` as an instance of the `Cat` class
- An object named `Duke` as an instance of the `Dog` class
- `speak(Mittens)`
  - "Meow!"
- `speak(Duke)`
  - "Woof!"

OOP in R versus other languages

In most object-oriented programming languages like C++ and Java, *methods belong to classes*. This relationship can be seen in the way they call their methods are called via `object.method()`:

- E.g., `Mittens.speak()`
- E.g., `Duke.speak()`

R takes a functional programming approach to OOP, so that *methods belong to generic functions*. This relationship can be seen in how methods in R are called via `method(object)`.

- E.g., `speak(Mittens)`
- E.g., `speak(Duke)`

This may seem confusing at first if you are familiar with OOP from a langauge like C++ or Java, but it's just a different way of thinking about OOP.

Object Systems in R

There are two major object-oriented programming systems in R:

- S3 classes:
  - Very simple class system
  - No formal class definitions
  - Single dispatch (methods only specialized on first argument)
- S4 classes:
  - More complex class system
  - Formal class definitions
  - Multiple dispatch (methods specialized on multiple arguments)

When to use which?

- Use S3 for simple data structures without complex dependencies
- Use S4 for more complex data structures

S3 is more common in base R and CRAN packages.

S4 is more common in Bioconductor packages.

Exceptions: RC

There is a third OOP system in R called Reference Classes, which we won't talk about in this class, because they break fundamental assumptions about data in R. They are useful, however, for classes which care about mutable state, such as GUIs.

The S3 OO System

The S3 class system is based on adding `attributes` to any of R's base types.

That means S3 classes are based on:

- `integer`
- `numeric`
- `character`
- `list`

...etc.

S3 classes are defined by their `class` attribute which can be accessed and set by the `class()` function.

What are some S3 classes you already know?

Existing S3 classes: factors

```
fc <- factor(c("a", "a", "b", "c"))
typeof(fc) # base type
```

```
## [1] "integer"
```

```
class(fc) # class
```

```
## [1] "factor"
```

```
attributes(fc)
```

```
## $levels
## [1] "a" "b" "c"
##
## $class
## [1] "factor"
```

Existing S3 classes: data.frames

```
df <- data.frame(x=1:3, y=4:6)
typeof(df) # base type
```

```
## [1] "list"
```

```
class(df) # class
```

```
## [1] "data.frame"
```

```
attributes(df)
```

```
## $names
## [1] "x" "y"
##
## $row.names
## [1] 1 2 3
##
## $class
## [1] "data.frame"
```

Existing S3 classes: `tibbles`

```
tb <- tibble(x=1:3, y=4:6)
typeof(tb) # base type
```

```
## [1] "list"
```

```
class(tb) # class - tbl_df is the actual class name
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
attributes(tb)
```

```
## $names
## [1] "x" "y"
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
##
## $row.names
## [1] 1 2 3
```

Existing S3 classes: `lm`

```
fit <- lm(y ~ x, data=sim1)
typeof(fit) # base type
```

```
## [1] "list"
```

```
class(fit) # class
```

```
## [1] "lm"
```

```
attributes(fit)
```

```
## $names
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
##
## $class
## [1] "lm"
```

S3 Generic Functions

S3 generic functions are defined by a call to `UseMethod()`.

Consider the following generic functions for extracting the fitted response values and residuals from a model.

```
fitted
```

```
## function (object, ...)
## UseMethod("fitted")
## <bytecode: 0x7fb2d66925f0>
## <environment: namespace:stats>
```

```
residuals
```

```
## function (object, ...)
## UseMethod("residuals")
## <bytecode: 0x7fb2d806de08>
## <environment: namespace:stats>
```

S3 Methods

We can view the methods defined for various classes using the `methods()` function.

```
methods(fitted)
```

```
##  [1] fitted.default*       fitted.gls*           fitted.glsStruct*
##  [4] fitted.gnls*          fitted.gnlsStruct*    fitted.isoreg*
##  [7] fitted.kmeans*        fitted.lme*           fitted.lmeStruct*
## [10] fitted.lmList*        fitted.nlmeStruct*    fitted.nls*
## [13] fitted.smooth.spline*
## see '?methods' for accessing help and source code
```

```
methods(residuals)
```

```
##  [1] residuals.default*       residuals.glm
##  [3] residuals.gls*           residuals.glsStruct*
##  [5] residuals.gnls*          residuals.gnlsStruct*
##  [7] residuals.HoltWinters*   residuals.isoreg*
##  [9] residuals.lm             residuals.lme*
## [11] residuals.lmeStruct*     residuals.lmList*
## [13] residuals.nlmeStruct*    residuals.nls*
## [15] residuals.psych*         residuals.smooth.spline*
## [17] residuals.tukeyline*
```

S3 Methods (cont'd)

S3 methods are defined by the naming convention `generic.class()`. S3 method dispatch actually relies this naming scheme, and there are no additional requirements for defining an S3 method for a particular class.

For `residuals`, we saw a `residuals.lm` method, but there was no `fitted.lm` method.

If there is no specific method for a class, the default method is called, as defined by a function called `generic.default()`.

S3 Methods (cont'd)

We can use `getS3method` to find a particular S3 method.

```
getS3method("fitted", "default")
```

```
## function (object, ...)
## {
##     xx <- if ("fitted.values" %in% names(object))
##         object$fitted.values
##     else object$fitted
##     napredict(object$na.action, xx)
## }
## <bytecode: 0x7fb2dac0fd50>
## <environment: namespace:stats>
```

S3 Methods (cont'd)

We can also view all the methods defined for a specific class using the `methods()` function.

```
methods(class="lm")
```

```
##  [1] add1             alias            anova            case.names
##  [5] coerce           confint          cooks.distance   deviance
##  [9] dfbeta           dfbetas          drop1            dummy.coef
## [13] effects          extractAIC       family           formula
## [17] fortify          hatvalues        influence        initialize
## [21] kappa            labels           logLik           model.frame
## [25] model.matrix     nobs             plot             predict
## [29] print            proj             qqnorm           qr
## [33] residuals        rstandard        rstudent         show
## [37] simulate         slotsFromS3      summary          variable.na
## [41] vcov
## see '?methods' for accessing help and source code
```

Defining an S3 class

We can define an S3 class either by using `structure()`, or by simply setting the `class` attribute of an existing object.

```
a <- structure(list(), class="Animal")
class(a)
```

```
## [1] "Animal"
```

```
a <- list()
class(a) <- "Animal"
class(a)
```

```
## [1] "Animal"
```

Create a constructor for an S3 class

Typically, we should provide a constructor for our class to make it easier to create an object of that class. Note we use S3 inheritance in this example.

```
Cat <- function(name) structure(list(name=name),
                                 class=c("Cat", "Animal"))
Dog <- function(name) structure(list(name=name),
                                 class=c("Dog", "Animal"))
Cat("Mittens")
```

```
## $name
## [1] "Mittens"
##
## attr(,"class")
## [1] "Cat"    "Animal"
```

```
Dog("Duke")
```

```
## $name
## [1] "Duke"
##
## attr(,"class")
## [1] "Dog"    "Animal"
```

Defining an S3 generic function

We now create a generic function for `speak` by creating a function that calls `UseMethod` with the name of our generic function.

```
speak <- function(object) UseMethod("speak")
```

Define a custom `print` method

We can create a custom print method for our classes by defining a `print` method. This is the generic function that gets called whenever we type the name of a variable and hit "Enter".

To do this, we simply define a function following the naming convention `generic.class()`. We'd like all animals to use the same `print` method, so we'll define it on `Animal`.

```
print.Animal <- function(object) {
  print(paste("A", class(object)[1] ,
              "named", object$name))
}
Cat("Mittens")
```

```
## [1] "A Cat named Mittens"
```

```
Dog("Duke")
```

```
## [1] "A Dog named Duke"
```

Defining S3 methods

We now create a `speak` methods for our classes by following the naming convention `generic.class()`.

```
speak.default <- function(object) print("*weird noises*")
speak.Animal <- function(object) print("*weird animal noises*")
speak.Cat <- function(object) print("Meow!")
speak.Dog <- function(object) print("Woof!")
```

S3 example

```
Mittens <- Cat("Mittens")
Duke <- Dog("Duke")
```

```
speak(Mittens)
```

```
## [1] "Meow!"
```

```
speak(Duke)
```

```
## [1] "Woof!"
```

```
speak(list())
```

```
## [1] "*weird noises*"
```

The S4 OO System

The S4 class system works similarly to S3 from a user perspective, for adds formality and rigor.

The S4 class system adds:

- Formal definitions of the data structure
  - S4 classes have `slots` (accessed via `@`) defined to be a specific data type
  - Inheritance is formally defined rather than via an attribute
- Method dispatch on multiple arguments, not only the first one
- Validity of the object can be rigorously checked

Creating an S4 class

S4 classes are defined via a call to `setClass`.

```
setClass("Animal4",
         contains = "VIRTUAL",
         slots = c(name = "character"),
         validity = function(object) {
           if ( length(object@name) != 1 )
             stop("slot 'name' must be length 1")
         })
setClass("Cat4", contains = "Animal4")
setClass("Dog4", contains = "Animal4")
```

We do not expect to actually create `Animal` objects, so we make it a `VIRTUAL` object. (Virtual classes cannot be instantiated.)

Create a constructor for an S4 class

Using `setClass` doesn't actually create or modify an existing object, so we should create constructors for our classes.

New instances of S4 classes are created using `new()`, but it is rude to ask the user to call `new()` directly.

```
Cat4 <- function(name) new("Cat4", name=name)
Dog4 <- function(name) new("Dog4", name=name)
Mittens4 <- Cat4("Mittens")
Duke4 <- Dog4("Duke")
Mittens4
```

```
## An object of class "Cat4"
## Slot "name":
## [1] "Mittens"
```

```
Duke4
```

```
## An object of class "Dog4"
## Slot "name":
## [1] "Duke"
```

Define a custom show method

S4 classes use the `show` generic function instead of the `print` generic function. S4 methods are defined using `setMethod`.

We'd like all animals to use the same `show` method, so we'll define it on `Animal4`.

```
setMethod("show", "Animal4", function(object) {
  print(paste("A", class(object)[1] ,
              "named", object@name))
})
```

```
## [1] "show"
```

```
Cat4("Mittens")
```

```
## [1] "A Cat4 named Mittens"
```

```
Dog4("Duke")
```

```
## [1] "A Dog4 named Duke"
```

Defining an S4 generic function

Just like S3 generic functions are defined by a call to `UseMethod()`, S4 generic functions are defined using `setGeneric()` with a call to `standardGeneric()`.

```
setGeneric("speak", function(object) standardGeneric("speak"))
```

```
## [1] "speak"
```

Defining S4 methods

Just like S3 generic functions are defined by a call to `UseMethod()`, S4 generic functions are defined using `setGeneric()` with a call to `standardGeneric()`.

```
setMethod("speak", "Cat4", function(object) print("Meow!"))
```

```
## [1] "speak"
```

```
setMethod("speak", "Dog4", function(object) print("Woof!"))
```

```
## [1] "speak"
```

Viewing existing S4 methods

We can view existing S4 methods with `showMethods()`.

```
showMethods("speak")
```

```
## Function: speak (package .GlobalEnv)
## object="ANY"
## object="Cat4"
## object="Dog4"
```

What is the method for class “ANY”?

Viewing existing S4 methods (cont'd)

We can view a specific method using `selectMethod()`.

```
selectMethod("speak", "ANY")
```

```
## Method Definition (Class "derivedDefaultMethod"):
##
## function (object)
## UseMethod("speak")
## <bytecode: 0x7fb2d8d5cf88>
##
## Signatures:
##         object
## target  "ANY"
## defined "ANY"
```

It's our S3 generic function!

S4 example

```
Mittens4 <- Cat4("Mittens")
Duke4 <- Dog4("Duke")
```

```
speak(Mittens4)
```

```
## [1] "Meow!"
```

```
speak(Duke4)
```

```
## [1] "Woof!"
```