# Functions and Functional Programming in R

Kylie Ariel Bemis

4 February 2020

# References for functions and OOP

*Advanced R* by Hadley Wickam

- `Book freely available at http://adv-r.had.co.nz`

*R Language Definition* by the R Core Team

- `https://cran.r-project.org/doc/manuals/R-lang.html`

# 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 an object that can be manipulated 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 R core 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`.

# Exercise: Write a function

Write a function that replaces all NAs in a numeric vector with the median value.

# Exercise: Write a function

```
impute_NA <- function(x) {
  ifelse(is.na(x), median(x, na.rm=TRUE), x)
}
```

# Why write a function?

Brainstorm some reasons you might want to write your own function.

# Why write a function?

- Avoid duplication
  - Copy-pasting code is BAD
  - Duplicated code introduces more possibility of errors
  - Difficult to change later
- Easily re-use a common workflow without copy-pasting code
- Share a new functionality with other people

# Which is better?

```
df$age <- ifelse(is.na(df$age), median(df$age, na.rm=TRUE), df$age)
df$time <- ifelse(is.na(df$time), median(df$sex, na.rm=TRUE), df$time)
df$year <- ifelse(is.na(df$year), median(df$year, na.rm=TRUE), df$year)
df$rate <- ifelse(is.na(df$rate), median(df$trt, na.rm=TRUE), df$rate)
```

vs

```
df$age <- impute_NA(df$age)
df$time <- impute_NA(df$time)
df$year <- impute_NA(df$year)
df$rate <- impute_NA(df$rate)
```

We'll see ways we can further improve on the second version later!

# A more flexible function

What if we want to allow the user to specify whether they want to impute using the median or the mean?

```
impute_NA2 <- function(x, method) {
  if ( method == "median" ) {
    ifelse(is.na(x), median(x, na.rm=TRUE), x)
  } else if ( method == "mean" ) {
    ifelse(is.na(x), mean(x, na.rm=TRUE), x)
  }
}

impute_NA2(c(1:3, NA, 5:9), "median")
```

```
## [1] 1.0 2.0 3.0 5.5 5.0 6.0 7.0 8.0 9.0
```

But now the user must always provide an extra argument!

# Default arguments

We can provide a default value that will be used for the `method` argument if none is provided by the user.

```
impute_NA3 <- function(x, method = "median") {
  if ( method == "median" ) {
    ifelse(is.na(x), median(x, na.rm=TRUE), x)
  } else if ( method == "mean" ) {
    ifelse(is.na(x), mean(x, na.rm=TRUE), x)
  }
}

impute_NA3(c(1:3, NA, 5:9))
```

```
## [1] 1.0 2.0 3.0 5.5 5.0 6.0 7.0 8.0 9.0
```

```
impute_NA3(c(1:3, NA, 5:9), "mean")
```

```
## [1] 1.000 2.000 3.000 5.125 5.000 6.000 7.000 8.000 9.000
```

# Using `match.arg()`

We can specify all possible values of `method` in the signature and use `match.arg()` to find the one that was provided.

```
impute_NA4 <- function(x, method = c("median", "mean")) {
  method <- match.arg(method)
  if ( method == "median" ) {
    ifelse(is.na(x), median(x, na.rm=TRUE), x)
  } else if ( method == "mean" ) {
    ifelse(is.na(x), mean(x, na.rm=TRUE), x)
  }
}
```

If no argument is provided, the first value will be used as the default.

```
impute_NA4(c(1:3, NA, 5:9))
```

```
## [1] 1.0 2.0 3.0 5.5 5.0 6.0 7.0 8.0 9.0
```

# Passing a function as an argument

What if we wanted to allow the user the specify the function used for imputation? Since functions are first-class citizens in R and can be passed around like any other object, we can allow a function as an argument.

```
impute_NA5 <- function(x, fun = median) {
  fun <- match.fun(fun)
  ifelse(is.na(x), fun(x, na.rm=TRUE), x)
}

impute_NA5(c(1:3, NA, 5:9))
```

```
## [1] 1.0 2.0 3.0 5.5 5.0 6.0 7.0 8.0 9.0
```

```
impute_NA5(c(1:3, NA, 5:9), mean)
```

```
## [1] 1.000 2.000 3.000 5.125 5.000 6.000 7.000 8.000 9.000
```

# Writing good functions

- Obviously correct
- Speed – correct now, optimize later
- General/complex utility vs. simply/specific utility
- Concise/clever vs. verbose and easy-to-understand
- Useful/simple name (often hardest part!)

# Anonymous functions

What does the following do?

```
impute_NA5(c(1:3, NA, 5:9),
           fun=function(x, na.rm=TRUE)
             sum(x, na.rm=na.rm) / length(x))
```

```
## [1] 1.000000 2.000000 3.000000 4.555556 5.000000 6.000000 7.0
## [9] 9.000000
```

# Anonymous functions

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`.

# Passing arguments to internal functions

What happens if we don't specify `na.rm=TRUE` in the anonymous function signature below?

```
impute_NA5(c(1:3, NA, 5:9),
           fun=function(x) sum(x) / length(x))
```

```
## Error in fun(x, na.rm = TRUE): unused argument (na.rm = TRUE)
```

What happened?

# Passing arguments to internal functions

In our function definition, we called `fun(x, na.rm=TRUE)`, so whatever function is passed MUST accept `na.rm` as an argument.

If we don't know what arguments a function inside another function might accept, and want to allow the user to pass any arguments along to it, we can use `...`

```
impute_NA6 <- function(x, fun = median, ...) {
  fun <- match.fun(fun)
  ifelse(is.na(x), fun(x, ...), x)
}

impute_NA6(c(1:3, NA, 5:9), fun=median, na.rm=TRUE)
```

```
## [1] 1.0 2.0 3.0 5.5 5.0 6.0 7.0 8.0 9.0
```

# Lazy evaluation

What if we want to allow the user to pass some constant value to use for the imputation, but fallback on using the median otherwise?

```
impute_NA7 <- function(x, value = default.value) {
  if ( missing(value) )
    default.value <- median(x, na.rm=TRUE)
  ifelse(is.na(x), value, x)
}

impute_NA7(c(1:3, NA, 5:9), -100)
```

```
## [1]    1    2    3 -100    5    6    7    8    9
```

```
impute_NA7(c(1:3, NA, 5:9))
```

```
## [1] 1.0 2.0 3.0 5.5 5.0 6.0 7.0 8.0 9.0
```

# Lazy evaluation

There's a lot to unpack here.

- Where is `default.value` defined?
- What does `missing()` do?

```
impute_NA7 <- function(x, value = default.value) {
  if ( missing(value) )
    default.value <- median(x, na.rm=TRUE)
  ifelse(is.na(x), value, x)
}
```

# Lazy evaluation

We use `missing()` to check whether the user supplied a value for `value`. If they didn't, we assign the median value to `default.value`.

```
impute_NA7 <- function(x, value = default.value) {
  if ( missing(value) )
    default.value <- median(x, na.rm=TRUE)
  ifelse(is.na(x), value, x)
}
```

Note that `default.value` is defined inside the function, but we are able to use it as a default value for the `value` argument anyway.

This is called **lazy evaluation**. R doesn't need to know the value of a parameter until it's actually used.

But how does R know where to find the value of `default.value`?

# 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: 0x7faf3bc02af8>
```

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**.

# Functional programming and Apply functions

The `*apply` family of functions are a particularly important pattern of **functionals** in R.

Rather than using `for` loops, it is common to use the `*apply` family of functions. These allow applying a function over each element of a vector.

- `lapply` always returns its results as a list.
- `sapply` is a variant of `lapply` that attempts to simplify its final result
- `vapply` is a variant of `lapply` that simplifies its result according to a template.

# Apply functions (cont'd)

`lapply` always returns its results as a list.

```
x <- list(1:3, 4:6, 7:9)
lapply(x, sum)
```

```
## [[1]]
## [1] 6
##
## [[2]]
## [1] 15
##
## [[3]]
## [1] 24
```

# Apply functions (cont'd)

`sapply` is a variant of `lapply` that attempts to simplify its final result into a homogenous vector, matrix, or array.

```
x <- list(1:3, 4:6, 7:9)
sapply(x, sum)
```

```
## [1]  6 15 24
```

# Apply functions (cont'd)

`vapply` is a variant of `lapply` that simplifies its result according to a user-supplied template.

```
x <- list(1:3, 4:6, 7:9)
vapply(x, sum, numeric(1))
```

```
## [1]  6 15 24
```

# Using apply functions

How can we use an apply function to further improve the following code?

```
df$age <- impute_NA(df$age)
df$sex <- impute_NA(df$time)
df$year <- impute_NA(df$year)
df$rate <- impute_NA(df$rate)
```

# Using apply functions

```
df$age <- impute_NA(df$age)
df$sex <- impute_NA(df$time)
df$year <- impute_NA(df$year)
df$rate <- impute_NA(df$rate)
```

vs.

```
lapply(df, impute_NA)
```

# Functionals and anonymous functions

Anonymous functions are especially powerful in conjunction with the `*apply` family of functions.

```
lapply(df, function(x)
  ifelse(is.na(x), median(x, na.rm=TRUE), x))
```

Sometimes using an anonymous function with an `*apply` function means you don't need to write a separate function in the first place!

# Variable number of arguments with ...

While `...` can be used to pass arguments to internal functions, it can also be used to write a function that can take a variable number of arguments.

```
imputeNAs <- function(...) {
  dots <- list(...)
  lapply(dots, function(x)
    ifelse(is.na(x), median(x, na.rm=TRUE), x))
}

imputeNAs(c(1:3, NA, 5:9), c(101:103, NA, 105:109))
```

```
## [[1]]
## [1] 1.0 2.0 3.0 5.5 5.0 6.0 7.0 8.0 9.0
##
## [[2]]
## [1] 101.0 102.0 103.0 105.5 105.0 106.0 107.0 108.0 109.0
```

# References

- http://adv-r.had.co.nz/Functions.html
- http://adv-r.had.co.nz/Functional-programming.html
- http://adv-r.had.co.nz/Functionals.html
- http://adv-r.had.co.nz/Function-operators.html