Programação em R

Julho, 2016

Funções: definição, argumentos e operadores binários

Por que funções?

Uma das grandes vantagens de usar uma linguagem de programação é automatizar o seu trabalho ou análise. Você será capaz de realizar grande parte do trabalho utilizando as funções internas do R ou de pacotes de terceiros em um script. Entretanto, você ganha ainda mais flexibilidade e agilidade criando suas próprias funções.
Vejamos um exemplo de *script*:

```
# queremos converter para numeric, retirar os dados discrepantes
# dividir por 1000 e arredondar o resultado
precos <- c("0.1", "1250.55", "2346.87", "3467.40", "10000000")
precos <- as.numeric(precos)
precos <- precos[!(precos < 1 | precos > 10000)]
precos <- precos/1000
precos <- round(precos)
precos
## [1] 1 2 3
```

Por que funções?

Nosso *script* faz o trabalho corretamente. Mas imagine que você queira realizar o mesmo procedimento com um vetor de preços diferente, digamos, `precos2`. Da forma como o seu código foi feito, você terá que **copiar e colar** os comandos e substituir os nomes.

```
# novo vetor de precos
precos2 <- c("0.0074", "5547.85", "2669.98", "8789.45", "150000000")
precos2 <- as.numeric(precos2)
precos2 <- precos2[!(precos2 < 1 | precos2 > 10000)]
precos2 <- precos2/1000
precos2 <- round(precos2)
precos2
## [1] 6 3 9
```

Note como isto é ineficiente. Além do trabalho de copiar e colar todo o seu código para cada análise diferente que você desejar fazer, você ainda estará sujeito a diversos erros operacionais, como esquecer de trocar um dos nomes ao copiar e colar em cada análise.

Por que funções?

O ideal, aqui, é criar uma ***função*** que realize este trabalho! Daqui a pouco veremos como fazer isso, primeiramente vejamos como definir uma função no R.

Definição

Uma função, no R, é definida da seguinte forma:

```
nomeDaFuncao <- function(arg1, arg2, arg3 = default3,  ...){
  # corpo da função: uma série de comados válidos.
  return(resultado) # opcional
}
```

- o comando `function()` diz para o R que você está definindo uma função.
- os valores dentro dos parênteses de `function()` são os argumentos (ou parâmetros) da função. Argumentos podem ter valores default, que são definidos com o sinal de igualdade (no caso `arg3` tem como default o valor `default3`). Existe um parâmetro coringa muito útil, o `...`, o qual veremos mais a frente.
- dentro das chaves encontra-se o "corpo" da função, isto é, uma série de comandos válidos que serão realizados.
- o comando `return()` encerra a função e retorna seu argumento. Como veremos, o `return()` é opcional. Caso omitido, a função retorna o último objeto calculado. ***A função só pode retornar um objeto***!

Definição

Criemos nossas primeiras funções:

```
quadrado <- function(x){
  x ^ 2
}
quadrado(3)
## [1] 9
## se for na mesma linha não precisa das chaves
quadrado <- function(x) x ^ 2
quadrado(3)
## [1] 9

## adicionando mais argumentos
elevado_n <- function(x,n) x ^ n
elevado_n(3, 3)
## [1] 27
```

Definição

Funções criam um ambiente local e, em geral, ***não alteram diretamente o objeto ao qual são aplicadas***. Elas tomam um objeto como argumento e criam outro objeto, modificado, como resultado. Na maior parte dos casos, a idéia é não ter efeitos colaterais com objetos fora da função.

```
x <- 10
elevado_n(x, 2) # isso altera o valor de x?
## [1] 100
x # não
## [1] 10

# se você quer salvar o resultado
# tem que atribuir a outro objeto
y <- elevado_n(x, 2)
y
## [1] 100
```

Exercícios

Sua vez.

- Crie uma função `soma_e_subtrai(x, y)` que receba um parâmetro `x` e outro `y` e retorne o resultado tanto da soma `x + y` quanto da subtração `x - y`. Lembre que uma função no R retorna apenas um objeto, você terá que combinar os resultados (em um vetor, por exemplo) antes de retorná-los.
- Crie os objetos `z <- 1` e `w <- 2`. Teste sua função usando `z` e `w`.

Exercícios

```
soma_e_subtrai <- function(x, y){
  soma <- x + y
  subtracao <- x - y
  resultado <- c(soma = soma, subtracao = subtracao)
  return(resultado)
}

z <- 1
w <- 2

soma_e_subtrai(z, w)
##      soma subtracao
##         3        -1
```

Voltando ao exemplo

Vamos voltar ao nosso exemplo inicial. Montemos uma função que realiza o tratamento dos dados visto anteriormente:

```
limparDados <- function(dados){
  dados <- as.numeric(dados)
  dados <- dados[!(dados < 1 | dados > 10000)]
  dados <- dados/1000
  dados <- round(dados)
  return(dados)
}
ls() # note que a função foi criada
##  [1] "elevado_n"      "limparDados"    "precos"         "precos2"
##  [5] "quadrado"       "soma_e_subtrai" "w"              "x"
##  [9] "y"              "z"
```

Voltando ao exemplo

Vejamos em detalhes:

- o comando `function()` diz para o R que você está definindo uma função.
- os valores dentro dos parênteses de `function()` são os argumentos da função. No nosso caso definimos um único argumento chamado `dados`.
- dentro das chaves encontra-se o "corpo" da função, isto é, as operações que serão realizadas. Neste caso, transformamos em `numeric`, retiramos dados menores que 1 e maiores do que 10000, dividimos por 1000 e arredondamos.
- a função `return()` encerra a função e retorna o vetor `dados` modificado.

Voltando ao exemplo

Pronta a função, sempre que você quiser realizar essas operações em um vetor diferente, basta utilizar `limparDados`.

```
precos3 <-  c("0.02", "4560.45", "1234.32", "7894.41", "12000000")
precos4 <- c("0.001", "1500000", "1200.9", "2000.2", "4520.5")
precos5 <- c("0.05", "1500000", "1000000", "7123.4", "9871.5")

# limpando os dados
limparDados(precos3)
## [1] 5 1 8
limparDados(precos4)
## [1] 1 2 5
limparDados(precos5)
## [1]  7 10
```

Voltando ao exemplo

Bem melhor do que o *script*.
Note que tínhamos 3 vetores diferentes e bastou chamar a função três vezes, ao invés de ter que copiar e colar três vezes o código. Note, também, que se houver algum erro, **temos que corrigir apenas na função**.
E podemos refinar ainda mais `limparDados`.
Por exemplo, da forma como a função está escrita, os valores de corte de mínimo e de máximo serão sempre 1 e 10000; além disso, os resultados sempre serão divididos por 1000.
E se quisermos modificar esses valores? Basta colocá-los também como argumentos.

Mais argumentos

Colocando mais argumentos:

```
limparDados <- function(dados, min, max, div){
  dados <- as.numeric(dados)
  dados <- dados[!(dados < min | dados > max)]
  dados <- dados/div
  dados <- round(dados)
  return(dados)
}
```

Agora você pode alterar os valores de `min`, `max` e `div` ao aplicar a função.

Mais argumentos

```
precos3 <-  c("0.02", "4560", "1234", "7894", "12000000")

limparDados(precos3, min = 0, max = 5000, div = 2)
## [1]    0 2280  617

limparDados(precos3, min = 0, max = 4000, div = 4)
## [1]   0 308

limparDados(precos3, min = -Inf, max = Inf, div = 1)
## [1] 0.0e+00 4.6e+03 1.2e+03 7.9e+03 1.2e+07
```

Mais argumentos

Note que argumentos são nomeados. Se você colocar os argumentos com seus nomes, a ordem dos argumentos não importa. Entretanto, você pode omitir os nomes dos argumentos, desde que os coloque em ordem correta.

```
#  argumentos em ordem diferente
limparDados(max = 5000, div = 2, min = 0, dados = precos3)
## [1]    0 2280  617

# argumentos sem nomes (na ordem)
limparDados(precos3, 0, 4000, 4)
## [1]   0 308
```

Argumentos Default

Com mais argumentos, se você esquecer de especificar algum, ocorrerá um erro:

```
limparDados(precos3, max = 5000, div = 1)
## Error in limparDados(precos3, max = 5000, div = 1): argumento "min" ausente, se
```

Para sanar isto, basta definir valores padrão (default).

Argumentos Default

Colocando argumentos padrão (default):

```
limparDados <- function(dados, min = 1, max = 10000, div = 1000){
  dados <- as.numeric(dados)
  dados <- dados[!(dados < min | dados > max)]
  dados <- dados/div
  dados <- round(dados)
  return(dados)
}
```

Podemos usar a função omitindo os argumentos que possuem default.

Argumentos Default

```
# usa o default para min
limparDados(precos3, max = 5000, div = 1)
## [1] 4560 1234

# usa o default para min e div
limparDados(precos3, max = Inf)
## [1]     5     1     8 12000

# usa o default para tudo
limparDados(precos3)
## [1] 5 1 8
```

Exercícios

Sua vez.

- Crie uma função `divide(x, divisor)` que divide `x` pelo `divisor` e retorne o resultado. Faça com que a função tenha um default de `divisor = 1` caso o usuário não passe nada. Teste sua função.
- Crie uma função `media(x, remove.nas)` que calcule a media de `x` usando a função `mean()` e que possa remover ou não os `NA`s dependendo do parâmetro `remove.nas` (lembre que a função `mean` já tem o argumento `na.rm` para remover `NA`s, você vai simplesmente repassar o argumento). Faça com que a função tenha como default `remove.nas = TRUE`. Teste sua função.

Exercícios

```
divide <- function(x, divisor = 1){
  resultado <- x/divisor
  return(resultado)
}

divide(10)
## [1] 10
divide(10, 5)
## [1] 2

media <- function(x, remove.nas = TRUE) mean(x, na.rm = remove.nas)
media(c(NA, 1:10))
## [1] 5.5
```

Funções podem ser argumentos

Funções também podem ser passadas como argumentos de funções. Por exemplo, suponha que você não queira sempre usar o `round()` para arredondamento. Você pode colocar a função final que é aplicada a `dados` como um argumento.

```
limparDados <- function(dados, min = 1,
                        max = 10000, div = 1000, funcao = round){
  dados <- as.numeric(dados)
  dados <- dados[!(dados < min | dados > max)]
  dados <- dados/div
  dados <- funcao(dados)
  return(dados)
}
```

Funções podem ser argumentos

Se quisermos usar a função `floor()` ao invés de `round()`, basta trocar o argumento `funcao`.

```
# usou os defaults
limparDados(precos3)
## [1] 5 1 8

# usa floor ao invés de round
limparDados(precos3, funcao = floor)
## [1] 4 1 7

# funcao anonima que pega x e retorna x (não faz nada com x)
limparDados(precos3, funcao = function(x) x)
## [1] 4.6 1.2 7.9
```

Funções anônimas

Como vimos no exemplo anterior, você pode definir uma função no próprio argumento. Estas funções são chamadas de **anônimas**.

```
limparDados(precos3, funcao = function(x) x)
## [1] 4.6 1.2 7.9
limparDados(precos3, funcao = function(x) x ^ 2)
## [1] 20.8  1.5 62.3
limparDados(precos3, funcao = function(x) log(x + 1))
## [1] 1.7 0.8 2.2
limparDados(precos3, funcao = function(x) {
  x <- round(x)
  x <- as.complex(x)
  x <- (-x) ^ (x/10)
} )
## [1]  0.00-2.24i  0.95-0.31i -4.27-3.10i
```

O ...

O R tem ainda um argumento *coringa* os "três pontos" `...`.
O `...` permite repassar argumentos para outras funções dentro da sua função, sem necessariamente ter que elencar todas as possibilidades de antemão.

```
limparDados <- function(dados, min = 1,
                        max = 10000, div = 1000, funcao = round, ...){
  dados <- as.numeric(dados)
  dados <- dados[!(dados < min | dados > max)]
  dados <- dados/div
  dados <- funcao(dados, ...)
  return(dados)
}
```

Agora podemos passar argumentos para `funcao(dados, ...)`

O ...

A função `round()`, por exemplo, tem o argumento `digits`. Ou a nossa função `elevado_n()` tem o argumento `n`. Podemos repassar esses argumentos para essas funções por meio do `...`.

```
limparDados(precos3)
## [1] 5 1 8
limparDados(precos3, digits = 1)
## [1] 4.6 1.2 7.9
limparDados(precos3, funcao = elevado_n, n = 2)
## [1] 20.8  1.5 62.3
```

Operadores binários

Lembra que falamos que +, no R, na verdade é a função `'+'(x,y)`? Funções deste tipo são chamadas de operadores binários. E, no R, você também pode definir seus operadores binários, com auxílio do %.
Vamos fazer um operador binário que cole textos:

```
"%+%" <- function(x, y) paste(x, y)
```

Agora podemos colar textos usando %+%:

```
"colando" %+% "textos" %+% "com nosso" %+% "operador"
## [1] "colando textos com nosso operador"
```

Operadores binários

Vejamos outro exemplo:

```
"%depois%" <- function(x, fun) fun(x)
```

Olhe que interessante:

```
set.seed(10)
x <- rnorm(100)
sqrt(exp(mean(x)))
## [1] 0.93
x %depois% mean %depois% exp %depois% sqrt
## [1] 0.93
```

A imaginação é o limite.

Exercícios

Sua vez.

- Defina uma função que retorne o mínimo, a mediana e o máximo de um vetor. Faça com que a função lide com NA's e que isso seja um argumento com default;
- Defina uma versão "operador binário" da função `rep`. Faça com que tenha seguinte sintaxe: `x %rep% n` retorna o objeto `x` repetido `n` vezes.
- Defina uma função que normalize/padronize um vetor (isto é, subtraia a média e divida pelo desvio-padrão). Faça com que a função tenha a opção de ignorar NA's. Permita ao usuário escolher outros parâmetros para a média (Dica: ...);
- Dados um vetor y e uma matriz de variáveis explicativas X, defina uma função que retorne os parâmetros de uma regressão linear de x contra y, juntamente com os dados originais usados na regressão. (Dicas: use álgebra matricial. Use uma lista para retornar o resultado)

Soluções

```
mmm <- function(x, na.rm = TRUE){

  # calcula min, median, e max, guarda em resultado
  resultado <- c(min(x, na.rm = na.rm),
                 median(x, na.rm = na.rm),
                 max(x, na.rm = na.rm))

  # nomeia o vetor para facilitar consulta
  names(resultado) <- c("min", "mediana", "max")

  #retorna vetor
  return(resultado)
}

mmm(c(1,2,3, NA))
##     min mediana     max
##       1       2       3
```

Soluções

```
"%rep%" <- function(x, n) rep(x, n)


7 %rep% 5
## [1] 7 7 7 7 7
```

Soluções

```r
padronize <- function(x, na.rm = TRUE, ...){

  m <- mean(x, na.rm = na.rm, ...) # calcule a média
  dp <- sd(x, na.rm = na.rm)       # calcule o dp
  pad <- (x - m)/dp                # padronize os dados

  attr(pad, "media") <- m        # guarda a média original p/ consulta
  attr(pad, "dp") <- dp          # guarda o dp original p/ consulta

  return(pad)                    # retorna o vetor pad já com atributos
}

padronize(1:5)
```

Soluções

```r
ols <- function(X, y){

  b <- solve(t(X) %*% X) %*% t(X) %*% y  # ols ((X'X)^-1)X'Y

  # guarda resultados em lista nomeada
  resultado <- list(coef = b, X = X, y = y)

  # retorna resultado
  return(resultado)
}
```

Soluções

Testando nosa função `ols()`:

```
# cria dados simulados
set.seed(10)                          # para reproducibilidade
X <- matrix(rnorm(300), ncol = 3)       # vetor X
y <- X %*% c(3,6,9) + rnorm(100)        # y = Xb + e, b=c(3, 6, 9), e~N(0,1)

# vamos testar a formula
resultado <- ols(X, y)
str(resultado)
resultado$coef
```

Um pouco sobre escopo de funções

Escopo

O R tem o que se chama, em programação, de **escopo léxico**. Grosso modo, quando uma função não encontra um objeto dentro de seu ambiente local, ela procura nos seus ambientes "pais" e, somente se não encontrar nada, retorna um erro.

```
x <- 2 # definimos um x na área de trabalho
func <- function(){
  x <- 3
  # definimos um x dentro da função
  # isso altera o valor de x fora da função?
  print(x)
  # print vai ser 2 ou 3?
  rm(x)
  # removemos o x. Qual x?
  print(x)
  # vai dar erro?
}
```

Escopo

Testando:

```
func()
## [1] 3
## [1] 2
```

Vamos remover x da área de trabalho.

```
rm(x)
```

E agora?

```
func()
## [1] 3
## Error in print(x): objeto 'x' não encontrado
```

Escopo

Vamos complicar o exemplo. Vamos definir uma função `func2()` que define um x localmente e chama a função `func()` definida anteriormente:

```
x <- 500
func2 <- function(){
 x <- 1000
 func() # vai achar quais x?
}

func2()
## [1] 3
## [1] 500
rm(x) # e agora?
func2()
## [1] 3
## Error in print(x): objeto 'x' não encontrado
```

Escopo

A função `func()` não encontra o `x` definido dentro do ambiente local de `func2()` porque o ambiente "pai" de `func()` é o ambiente global.
Não entraremos em detalhes de escopo léxico vs escopo dinâmico, mas é importante você entender um pouco disto pois alguns "bugs" que você irá encontrar na verdade são comportamentos esperados.

Um pouco sobre métodos de funções

Métodos

Um último assunto que veremos brevemente é o uso de métodos em funções no R. Para ilustrar, vejamos a definição da função `plot()`.

```
plot
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x7fa5623b7a70>
## <environment: namespace:graphics>
```

O único comando da função é `UseMethod("plot")`!
O que isso quer dizer?

Métodos

Na verdade, a função `plot()` é uma **função genérica**.
Ela verifica a classe do objeto em que está sendo aplicada e repassa os seus argumentos `(x, y, ...)` para funções específicas de `plot` a depender da classe do objeto. Essas funções específicas, ou métodos, são denominadas `plot.classe.do.objeto`. Se não tiver um método específico para a classe, os argumentos são repassados para a função default (`plot.default`).
Digite os seguintes comandos para ver as funções `plot.default()` e `plot.data.frame()`:

```
plot.default
graphics:::plot.data.frame
```

O mesmo ocorre com a fução `print()`.

Métodos

Para exemplificar, vamos complementar a função `ols()` que criamos anteriormente, e definir que o objeto que ela retorna é de classe "nossa_classe".

```
ols <- function(X, y){
  b <- solve(t(X) %*% X) %*% t(X) %*% y
  resultado <- list(coef = b, X = X, y = y)
  class(resultado) <- "nossa_classe"
  return(resultado)
}
```

Métodos

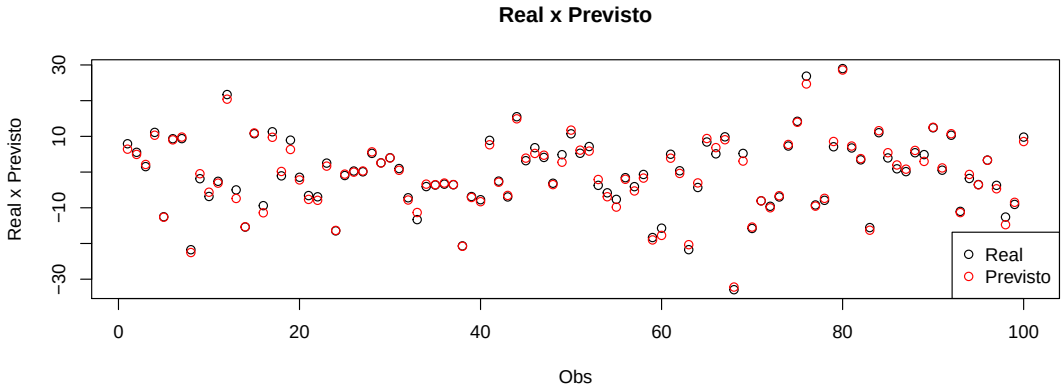
Vamos definir um método de `plot` para a classe "nossa_classe".

```
plot.nossa_classe <- function(x) {
  previsto <- X %*% (x$coef)
  plot(y,
       xlab = "Obs",
       main = "Real x Previsto",
       ylab = "Real x Previsto")
  points(previsto, col = "red")
  legend("bottomright", col = c("black", "red"),
         pch=1, legend = c("Real", "Previsto")
         )
}
```

Quando você chamar a função `plot()` para o resultado da função `ols`, ela automaticamente repassará os argumentos para a função `plot.nossa_classe()`.

Métodos

```
resultado <- ols(X, y)
plot(resultado)
```



Métodos

Outro exemplo: vamos criar métodos para nossa função `limparDados`. Note que a função dá erro se o argumento `dados` for um `data.frame`.

```
df_precos <- data.frame(precos3,
                        precos4,
                        precos5,
                        stringsAsFactors = FALSE)
# vai dar erro
limparDados(df_precos)
## Error in limparDados(df_precos): objeto (list) não é coercível para tipo 'doubl
```

Métodos

Vamos, então, definir a função que criamos como a função default e definir `limparDados` como uma função genérica.

```
# copiando limparDados para limparDados.default
limparDados.default <- limparDados

# definindo limparDados como genérica
limparDados <- function(dados, ...){
  UseMethod("limparDados")
}
```

Métodos

Note que `limparDados` continua funcionando normalmente com os vetores simples.

```
limparDados(precos3)
## [1] 5 1 8
```

Vamos criar um método de `data.frame` para `limparDados` (neste caso vamos usar `lapply()` para aplicar a função em todas as colunas do data.frame):

```
limparDados.data.frame <- function(dados, ...){
  lapply(dados, limparDados, ...)
}
```

Métodos

Agora a função automaticamente procura o método adequado para a classe de objeto ao qual está sendo aplicada.

```
limparDados(precos3)
## [1] 5 1 8
limparDados(df_precos)
## $precos3
## [1] 5 1 8
##
## $precos4
## [1] 1 2 5
##
## $precos5
## [1]  7 10
```

Métodos

Para ver os métodos de uma função, use `methods(funcao)`.

```
# Irá mostrar todos os métodos de print
# dos pacotes carregados
methods(print)

# Irá mostrar todos os métodos de plot
# dos pacotes carregados
methods(plot)
```

Para ver os métodos de uma classe, use `methods(class="classe")`.

```
# Irá mostrar todos os métodos para data.frame
methods(class="data.frame")
```