Estatística Computacional I

Lupércio França Bessegato
Dep. de Estatística/UFJF

Roteiro Geral

1. Programando em R
2. Preparação, limpeza e manipulação de dados
3. Gráficos em R
4. Tópicos especiais
5. Referências

Programando em R

Codificação

Debbuging

- Em geral, cometemos erros na construção de códigos com muitas funções
- Exemplos:
  - √ Mensagem de erro

```
> "a" + 1
Error in "a" + 1 : argumento não-numérico para operador binário
```

  - √ Aviso:

```
> 1:3 + 1:2
[1] 2 4 4
Warning message:
In 1:3 + 1:2 :
  comprimento do objeto maior não é múltiplo do comprimento do objeto menor
```

    - Não ignore!
      - Sinal de que algo não funciona como pretendido

Prevenção de Erros

- Construa seu código em módulos
  - √ Não se repita
    - Crie funções para executar tarefas específicas e as execute quando necessário
    - Use o mesmo código, caso necessite realizar a mesma tarefa várias vezes
    - É mais fácil detectar e corrigir quaisquer erros.

- Sempre que possível escreva seu código de maneira simplificada
  - √ Aproveite os recursos incorporados no R que permitem que códigos complicados sejam escritos de maneira muito sucinta e clara
    - Ex.: família `apply`
  - √ Aprenda esses recursos e tente aplicá-los sempre

- Comente seu código
  - √ Para qualquer pessoa entender o que ele pretende fazer
    - Muitas vezes será você quem se esquecerá do que estava fazendo quando o escreveu
  - √ Dê nomes descritivos para as variáveis e funções
    - Contextualizados ao problema

- Pense sempre nos casos especiais
  - √ Por exemplo, um erro comum é não pensar sobre o que acontecerá quando sua entrada for de comprimento 1 ou 0

```
>   # Função para somar subconjunto de colunas de uma matriz
> soma.linhas <- function(x, cols){
+ apply(x[, cols], 1, sum)
+ }
> (x <- matrix(1:9, ncol = 3))
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> soma.linhas(x, 2:3)
[1] 11 13 15
> soma.linhas(x, 2)]
Erro: ']' inesperado in "soma.linhas(x, 2)]"
> dim(x[, 2])
NULL
> is.vector(x[, 2])
[1] TRUE
```


- √ Solução:
  - `drop = FALSE`
    - As dimensões do objeto são mantidas

```
> # solução - argumento drop
> soma.linhas <- function(x, cols){
+ apply(x[, cols, drop = FALSE], 1, sum)
+ }
> soma.linhas(x, 2)
[1] 4 5 6
> soma.linhas(x, 2:3)
[1] 11 13 15
> x[, 2]
[1] 4 5 6
> x[, 2, drop = FALSE]
     [,1]
[1,]    4
[2,]    5
[3,]    6
```


- Codificação em módulos:
  - √ R é uma linguagem funcional
    - As operações são realizadas por funções, que geralmente são independentes
  - √ É boa prática manter seu código modular:
    - Ser composto de funções distintas que executam pequenas tarefas
    - É possível testá-las (e depurá-las) individualmente
  - √ Instrução complexa que não funciona:
    - Divida-a em partes e verifique se cada uma delas está fazendo o que você pretende

- √ Exemplo:

```
>   # modularidade
> x <- matrix(1:9, ncol = 3)
> y <- NULL
> if(any(x > 3) && y != 2){
+ print("Passei por aqui")
+ }
Error in if (any(x > 3) && y != 2) { :
  valor ausente onde TRUE/FALSE necessário
> any(x > 3)
[1] TRUE
> y != 2
logical(0)
> y
NULL
```

√ Redução da complexidade de códigos mais complicados:

- Escrever funções separadas, responsáveis por parcelas de seu programa
- Testá-las separadamente com relação a resultados e erros Estes podem então ser testados para erros separadamente
- Facilita sua reutilização em outras circunstâncias

√ Exemplo:

```
>   # Exemplo
> g <- function(y){
+ if (y < 0) warning("Aviso")
+ return(y)
+ }
> h <- function(z){
+ stop("Mensagem de erro")
+ z
+ }
> f <- function(x){
+ # função que executa outras funções
+ saida = g(x) + h(x)
+ }
> f(2)
Error in h(x) : Mensagem de erro
```


√ É relativamente fácil rastrear erros até a função específica na qual eles ocorrem

- Mais difícil procurar problemas no código

```
> traceback()
3: stop("Mensagem de erro") at #2
2: h(x) at #3
1: f(2)
```

- Pode-se corrigir a função h

```
>   # correção da função h
> h = function(z){
+ z
+ }
> f(-2)
Warning message:
In g(x) : Aviso
```

- `traceback` não disponível para `warnings`

- Pode-se transformar `warning` em `error`, com o propósito de debugar a função

```
> options(warn = 2)
> f(-2)
Error in g(x) : (convertido do aviso) Aviso
> traceback()
7: doWithOneRestart(return(expr), restart)
6: withOneRestart(expr, restarts[[1L]])
5: withRestarts({
       .Internal(.signalCondition(simpleWarning(msg, call), msg,
           call))
       .Internal(.dfltWarn(msg, call))
   }, muffleWarning = function() NULL)
4: .signalSimpleWarning("Aviso", quote(g(x)))
3: warning("Aviso") at #2
2: g(x) at #3
1: f(-2)
```

- Recomendável inserção de `stop` na função `g`

√ `options(error = recover)`
- Se ocorrer erro durante execução de função, serão mostradas as linhas executadas
- Escolhida a linha, pode-se inspecionar objetos dentro do comando selecionado
- Digite `Q` para sair do browser

√ `options(error = NULL)`
- Retorna ao modo normal de trabalho

√ Execução:

```
>   # Recuperação de erro
> options(error = recover)
> f(-2)
Error in g(x) : (convertido do aviso) Aviso

Enter a frame number, or 0 to exit

1: f(-2)
2: #3: g(x)
3: #2: warning("Aviso")
4: .signalSimpleWarning("Aviso", quote(g(x)))
5: withRestarts({
    .Internal(.signalCondition(simpleWarning(msg, call), msg
6: withOneRestart(expr, restarts[[1]])
7: doWithOneRestart(return(expr), restart)
Selection: 3
Called from: withOneRestart(expr, restarts[[1L]])
Browse[1]> stop("Escolha novo valor, que deve ser >=0")
Erros durante o embrulho: Escolha novo valor, que deve ser >=0
Browse[1]> Q
>
> options(error = NULL)
```


√ Comando `debug`:
- Inspeciona a execução de cada linha da função
- Digitando ENTER se passa de linha em linha
- Digite `Q` para sair do browser

√ `Comando undebug`
- Retorna ao modo normal de trabalho

```
> debug(g)
> f(2)
debugging in: g(x)
debug em <tmp>#1: {
    if (y < 0)
        warning("Aviso")
    return(y)
}
Browse[2]>
debug em <tmp>#2: if (y < 0) warning("Aviso")
Browse[2]>
debug em <tmp>#3: return(y)
Browse[2]> Q
> undebug(g)
```


√ Comando `trace`:
- Permite edição temporária da função
  - Alterações não são permanentes
- Use print para facilitar a verificação

√ `Comando untrace`
- Reverte para a função original

```
> untrace(g)
>   # Comando trace
> trace(g, edit = TRUE)
[1] "g"
> untrace(g)
```

Referências

Bibliografia Recomendada

- ALBERT, J.; RIZZO, M. *R by Example*. Springer, 2012.
- CHRISTIAN, N. *Basic Programming*, Lecture Notes
- DALGAARD, P. *Introductory statistics with R*. Springer, 2008.
- KLEIBER, C.; ZEILEIS, A. *Applied econometrics with R*. Springer, 2008.
- GARDENER, M. *Beginning R: The statistical programming language*. John Wiley & Sons, 2012.