Matéria de Análise de Dados – Ciências Biomédicas

Aula 13 – Programação em R - Conceitos Básicos

James R. Hunter

31 de março de 2017

Esta aula tratará alguns conceitos básicos de programação. Vocês estão aprendendo estatística mas aquela parte do curso que envolve como programar os comandos, carregar os dados e executar as análises precisa atenção. O objetivo aqui é de remover um pouco do mistério de como escrever programas em qualquer linguagem de informática, mas especialmente em nosso caso em R.

Já estudamos algumas aspetos importantes de programação:

- Como carregar dados dos arquivos externos em R
  - com as funções do pacote `readr` como 'read_csv()
  - com `load` do R base
- Como manusear dados com as funções de `dplyr`

Aqui, vou apresentar conceitos e técnicas de fluxo de controle nas programas e como fazer testes condicionais. Também, olhamos em algoritmos como ordenação e classificação dos dados. Lembramos que computação é basicamente o processo de começar com alguns dados como inputs, aplicar a eles um algoritmo e mostrar na tela, num arquivo ou impresso um output desse algoritmo.

Figure 1: Algoritmo

Fluxo de Control (*Control Flow*)

Um programa vai da primeira linha até a última sem exceção se o programa não contem alguma linha que transfere controle para 1) repete alguns comandos num loop ou 2) um outro lugar condicionalmente. Todos os algoritmos têm esses tipos de controles. Por exemplo, aqui é um pequeno teste condicional da função `lm` que executa modelos lineares em base R:

```
if (!is.null(w) && !is.numeric(w))
        stop("'weights' must be a numeric vector")
```

`for` Loops (*aka* Laços)

Loops permitem que um program pode fazer um tarifa repetitivamente. O sistema R tem vários tipos de loops, mas o tipo mais básico, um `for` loop seria a única que você precisa usar.

Um loop tem três elementos:

- a palavra `for`
- uma variável iteradora que assumirá valores sucessivos dos elementos de um objeto
- código a ser executado dentro dos chaves (`{}`)

O loop vai fazer iterações com a variável iteradora assumindo todos os valores indicados e executar o bloco de código entre os chaves.

Um dos usos principais para um loop é o cálculo dos valores de um `data.frame`, `tibble`, vetor ou outra estrutura dos dados baseado nos valores da variável iteradora.

Primeiro Exemplo

Um exemplo trivial, mas fácil a entender: queremos preencher os valores de uma variável `x` com os valores 1 até 6.

Podemos criar o seguinte loop:

```
x <- numeric(length = 6)  # inicializar o vetor e reservar a memoria necessária
for (i in seq(1, 6)) {
  x[i] <- i
}
x
```

```
## [1] 1 2 3 4 5 6
```

Aqui `i` é a variável iteradora, que vai assumir os valores 1 até 6 (dado pelo comando `seq` – *sequência*). O corpo do loop vai colocar o valor de `i` em cada elemento da variável `x` pelo assignment. `x` será preenchida assim:

- Depois de inicialização de x
  - x = [0 0 0 0 0 0]
- 1ª vez no loop: i = 1 e x[i] terá o valor 1 pelo assignment
  - x = [1 0 0 0 0 0]
- 2ª vez no loop: i = 2 e x[i] terá o valor 2 pelo assignment
  - x = [1 2 0 0 0 0]
- 3ª vez no loop: i = 3 e x[i] terá o valor 3 pelo assignment
  - x = [1 2 3 0 0 0]

e esse padrão continua até R completa a sexta rodada do loop.

Segundo Exemplo

Vamos agora criar um vetor dos valores aleatórios e vamos calcular o raiz quadrado desses números num loop. Nós vamos até começar com um número aleatório dos valores. Assim, não sabemos antes de executar o código quantas vezes o loop deve rodar. Para determinar quantas vezes a variável iterador deve incrementar, podemos usar a função `seq_along` que mede o tamanho de um vetor. Calculamos os números aleatórios usando a distribuição Uniforme, que dá para todos os números entre o mínimo e o máximo uma probabilidade igual de ser selecionado.

Anote que `seq_along` não precisa a parte "1:" da especificação do vetor porque faz si mesmo a sequência inteira dos elementos do objeto.

```
set.seed = 1946
trials <- floor(runif(n = 1, min = 1, max = 50))
numeros <-  runif(trials, min = 1, max = 100)
quads <- numeric(length = length(numeros))
for (i in seq_along(numeros)) {
```

```
  quads[i] <- sqrt(numeros[i])
}
quads[1:10] # Mostre primeiro 10 raizes quadrados
```

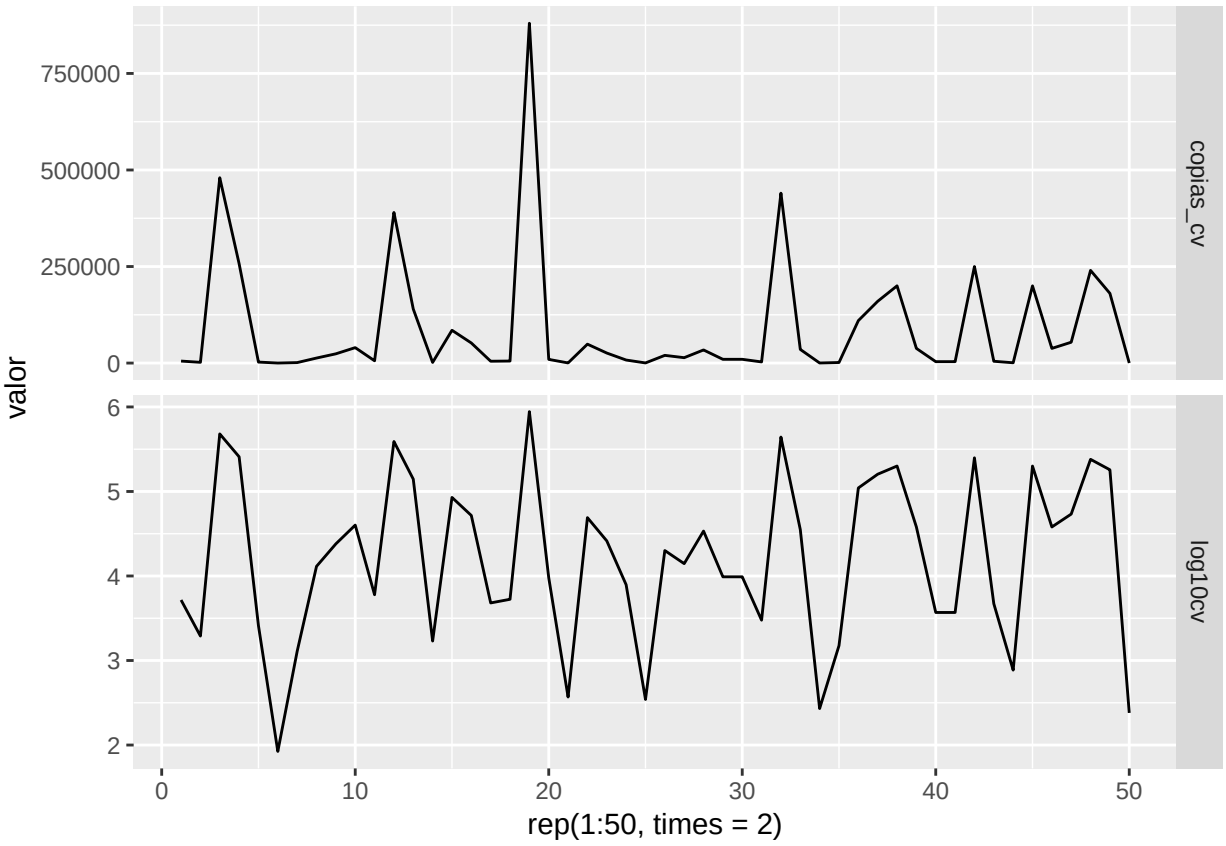
```
##  [1] 4.685958 6.705896 7.914399 4.818905 3.861873       NA       NA
##  [8]       NA       NA       NA
```

Agora, vamos trabalhar com uma amostra das cargas virais de pacientes de HIV e queremos criar uma variável no tibble com o logaritmo da carga viral. Já trabalhamos com esses dados do arquivo `counts_demo.csv`. Antes de começar o loop, precisamos carregar os dados.

```
cvdados <- read.csv("counts_demo.csv")
cvdados$log10cv <- rep(0, nrow(cvdados)) # inicializar nova variável
for (i in 1:nrow(cvdados)) {
  cvdados$log10cv[i] <- log10(cvdados$copias_cv[i])
}
kable(cvdados[1:10, c(1,4)])
```

copias_cv	log10cv
5200	3.716003
1947	3.289366
480000	5.681241
257313	5.410462
2585	3.412460
84	1.924279
1286	3.109241
13000	4.113943
24000	4.380211
40000	4.602060

```
grcv <- cvdados %>%
        select(copias_cv, log10cv) %>%
        gather(tipo, valor) %>%
        ggplot(aes(x = rep(1:50, times = 2), y = valor)) + geom_line() + facet_grid(tipo ~ ., scales = 
grcv
```



Loops Aninhados

Você pode aninhar loops, ou seja, fazer um loop dentro de um outro. O uso típico dessa estrutura é quando você trabalha com tipos das estruturas de dados que estão hierárquicas, como matrizes ou listas dentro de listas. Uma recomendação é de não fazer mais de dois ou três loops aninhados porque depois disso, fica muito difícil entender o código. Se você precisa fazer isso, ou pode dividir a operação em um número de operações separadas ou usar funções para fazer os loops interiores.

Vamos olhar rapidamente num exemplo pequeno de dois loops aninhados que vão imprimir os valores de um data frame.

```
X <- data.frame(a = c("a", "b", "c"), b = c("d", "e", "f"),
                stringsAsFactors = FALSE)
X
```

```
##   a b
## 1 a d
## 2 b e
## 3 c f
```

```
# loop
for (i in seq_len(ncol(X))) {
  for (j in seq_len(nrow(X))) {
    print(X[j, i])
  }
}
```

```
## [1] "a"
```

```
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
## [1] "f"
```

O programa escolheu a primeiro coluna (a) e imprimiu todas as linhas desta coluna. Depois mudou para segunda coluna e imprimiu as três linhas desta coluna (b).

Vetorização

R é uma linguagem *vetorizada*, que quer dizer que muitas operações podem ser feito em paralelo. Com vetorização, podemos especificar os valores de um vetor diretamente invés de fazer um loop como fizemos no primeiro exemplo acima.

O método vetorizado:

```
x <- 1:6
x
```

```
## [1] 1 2 3 4 5 6
```

Também podemos fazer operações aritméticas nos vetores diretamente sem recurso a um loop como neste exemplo:

```
x <- 1:4
y <- 4:1 # sequências podem ir para baixo também
z <- x + y
x; y; z
```

```
## [1] 1 2 3 4
```

```
## [1] 4 3 2 1
```

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

Além de adição, subtração, multiplicação e divisão são vetorizadas, tanto quanto comparações lógicas e operações com matrizes (includindo operações baseadas em elementos e verdadeiro operações matriciais – usando %*% e outros símbolos para as matrizes).

Funções

> Functions are used to **encapsulate** a sequences of expressions that are executed together to achieve a specific goal. (Peng, Kross & Anderson, *Mastering Software Development in R*, 2017, p. 99.)

Todos os comandos de R são funções. Mas, as vezes, precisamos fazer alguma tarifa repetitiva que os autores dos pacotes que usamos não desenharam. Invés de re-escrever o mesmo código cada vez precisamos ele, podemos colocar ele numa função. Quem utiliza o programa pode chamar ela a vontade.

Funções geralmente têm argumentos que o usuário pode modificar. Na aula sobre regressão logística, introduzimos uma função que calculou o inverso de um valor na escala de transformação `logit`. A função é simples. Ela manipula um valor numérico que o usuário dá para ela (x) e retorna o invlogit deste número. Dentro da função, a variável `x` usa o argumento para fazer os cálculos.

```
invlogit <- function(x) {
   1/(1 + exp(-x))
```

```
}
invlogit(10)
```

```
## [1] 0.9999546
```

Funções podem ser muito mais complicadas, mas, resumindo, elas servem para executar código que o programador quer usar mais que uma vez. Existem livros sobre o assunto de funções e não são o objetivo primário deste resumo de programação.

Família `apply` de Funções

Base R contêm uma serie de funções que ajuda com a aplicação dos comandos aos vetores ou outro objetos. Eles têm o nome `xapply()` onde o x pode ser trocado por um "l", "s", "t" ou "v". Vou apresentar aqui a versão chamada *simples* – `sapply`.

`sapply` aplica uma função a cada elemento de um vetor e retorna ou um vetor ou matriz. `replicate`, uma função que já usamos na aula é um membro de mesma família das funções. A sintaxe de `sapply` é fácil de entender. Você especifica o vetor (ou lista) e uma função a ser aplicada a ele. Alguns exemplos:

```
## Aplicada a um vetor
sapply(1:10, sqrt)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

```
## Aplicada a um data frame
sapply(cvdados, summary)
```

```
##         copias_cv contagem_cd4 contagem_cd8 log10cv
## Min.           84         66.0        393.0   1.924
## 1st Qu.      3700        250.8        652.0   3.568
## Median      17000        472.5        827.5   4.224
## Mean        90690        513.9        994.3   4.191
## 3rd Qu.    103800        676.0       1230.0   5.013
## Max.       880000       1603.0       2000.0   5.944
```

Dentro de `dplyr`, `mutate`, `transmute` e `summarize` podem conseguir o mesmo resultados para data frames ou tibbles. Entretanto, o "old-fashioned" `sapply` é frequentemente mais rápido para escrever, especialmente quando você está fazendo uma análise exploratória. Aqui é um exemplo em que `sapply` fica dentro do de uma sequência de comandos de *tidyverse*. No exemplo, as funções de `dplyr` seleciona os primeiros 10 casos da variável `log10cv` e a função `sapply` calcula o valor original para esses elementos da variável.

```
cvdados %>%
  select(log10cv) %>%
  slice(1:10) %>%
  sapply(function(x){10^x})
```

```
##       log10cv
##  [1,]    5200
##  [2,]    1947
##  [3,]  480000
##  [4,]  257313
##  [5,]    2585
##  [6,]      84
##  [7,]    1286
##  [8,]   13000
##  [9,]   24000
```

```
## [10,]   40000
```

Funções `map` no Pacote `purrr`

Parte do *tidyverse*, `purrr` tem uma família de funções chamada `map` que também segue este modelo de aplicar uma função a uma estrutura dos dados. `map` brinca educadamente com o *tidyverse*.

A sintaxe da família `map` é muito direto: `map(.x, .f, xxx)`. Para cada elemento de `.x` faça `.f`. `.x` pode ser um data frame, uma lista ou um vetor. `.x` e `.f` são marcadores para objetos e funções que você vai chamar, respectivamente.

A função `map` – sem sufixo – retornará uma lista ou a mesma classe de dados que ela recebeu do usuário. Existem outros tipos de `map` para classes diferentes de vetores:

- 'map_lgl() – vetores lógicos
- 'map_chr() – vetores de caráteres
- 'map_dbl() – vetores de números ("dbl" quer dizer dupla precisão)
- 'map_int() – vetores de números inteiros

Ao início, recomendo que vocês evitam a `map` que produz listas porque vocês não têm muito prática com listas. Invés pode usar calmamente os outros quatro tipos.

Alguns exemplos:

```
## Mesmo sintaxe que o exemplo de sapply
map_dbl(1:10, sqrt)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

```
## Aqui a função é uma função criado para este comando
map_lgl(c(1, 10, 4, 25, 100, 2000, 3, 5000), function(x) {
  x < 50
})
```

```
## [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE
```

Testes lógicos em R e qualquer outra linguagem permitem que você testar uma condição e fazer uma coisa se a condição for verdade ou outra coisa se for falso. Vamos começar com a versão mais simples deste tipo de estrutura de controle.

`if`

Usando só um `if` teste a condição e se for verdade, executa o código que você escreveu. Se for falso, o programa não faz nada e vai diretamente para a próxima linha de código.

```
if (condição) { fazer algo } ## continuar com o progama
```

Um exemplo. Vou executar duas vezes: uma em que a resposta para o teste é verdade e outro falsa.

```
## Caso verdadeiro
x <- 4
y <- "FALSO"
if(x == 4){
  y <-  "VERDADEIRO"
}
paste("Caso:", y)
```

```
## [1] "Caso: VERDADEIRO"
```

```
## Caso falso
x <- 4
y <- "FALSO"
if(x == 5){ ## Resultado falso
  y <-  "VERDADEIRO"
}
paste("Caso:", y)
```

```
## [1] "Caso: FALSO"
```

Anote que no segundo caso, `y` reteve o valor original que demos a ela porque o programa não fez nada para alterar o valor.

`if ... else`

O próximo nível de complexidade é para introduzir uma ação para condição negativa do teste utilizando o parâmetro `else`.

A sintaxe deste tipo de teste é:

```
if(condição) {
  ### fazer algo
} else {
  ### fazer outra coisa
}
```

Por exemplo, usando o modelo do exemplo acima.

```
## Caso verdadeiro
x <- 4
if(x == 4) {
  y = "VERDADEIRO"
} else {
  y <- "FALSO"
}
paste("Caso:", y)
```

```
## [1] "Caso: VERDADEIRO"
```

```
## Caso falso
x <- 4
if(x == 5){ ## Resultado falso
  y = "VERDADEIRO"
} else {
  y <- "FALSO"
}
paste("Caso:", y)
```

```
## [1] "Caso: FALSO"
```

Você pode aninhar múltiplas níveis de testes `if ... else` a vontade. Mas, a mesma advertência que fiz sobre loops deve ser respeitada: tente de ficar com um ou dois testes no mesmo bloco de código porque muitos testes podem confunde você e esconde o resultado.

Função `ifelse()`

Base R tem uma função que permite que você expressa todo o teste lógico dentro de um comando. A sintaxe fica simples: `ifelse(teste, sim, não)`.

Os três elementos do comando são

- `teste` – teste lógico
- `sim` – resultado para resposta VERDADEIRO
- `não` – resultado para resposta FALSA

Usando o mesmo exemplo de novo, temos:

```
x = 4

## Caso Verdadeiro
y <- ifelse(x == 4, y <- "VERDADEIRO", "FALSO")
paste("Caso:", y)
```

```
## [1] "Caso: VERDADEIRO"
```

```
## Caso Falso
y = ifelse(x == 5, "VERDADEIRO", "FALSO")
paste("Caso:", y)
```

```
## [1] "Caso: FALSO"
```

`ifelse` também é muito mais compacto em termos das linhas de código que precisa escrever.

Conclusão

Este é só uma introdução breve para os conceitos de controle de fluxo de um programa em R. Pode aprofundar muito mais, mas esta servirá para vocês possam iniciar programas um pouco mais sofisticados.