# Aula 5 - Laços, controladores e funções no R

Eduardo Koerich Nery

A complexidade do nosso código provavelmente vai aumentar a medida que desenvolvemos nossos projetos. Talvez precisaremos executar tarefas de forma repetida, logo precisaremos aplicar *laços* no nosso código. Talvez precisaremos executar tarefas apenas quando certos critérios são atendidos, logo precisaremos aplicar *controladores de fluxo* no nosso código. Por fim, podemos precisar executar uma tarefa que não está implementada em nenhuma biblioteca, logo precisaremos *definir funções* customizadas. Essas habilidades são a base da programação em R.

Nós usaremos os dados `Galton_dados_pt.csv` para desenvolver os tópicos desta aula. Importe os dados para o R e guarde os mesmos em um objeto chamado `galton`. Inspecione os dados para se familiarizar. Dado isto, vamos lá!

### Laços determinados (*for loops*)

Laços determinados permitem repetir tarefas por um número pré-determinado de vezes. Esses laços precisam de um objeto que mudará de valor a cada repetição, *o index*, e um vetor definindo os valores desse objeto para cada repetição. A estrutura geral dos laços determinados é:

```
for (*index* in *vetor com valores*){
  *tarefa a ser repetida*
}
```

Enquanto o index tiver valores para assumir, a tarefa continuará a ser repetida. Os valores que o index pode assumir são definidos no vetor após o operador `in`. As chaves `{}` delimitam a tarefa a ser repetida. A indentação (recúo de linha) não é necessária na linguagem R, mas é necessária em outras linguagens de programação. Contudo, a boa prática recomenda sinalizar os laços com identação para melhorar a leitura do código para você e, possivelmente, para outros usuários.

Por exemplo, se quisermos saber a diferença de altura entre cada criança e seu pai, podemos utilizar um laço determinado. Eu consigo saber quantas crianças foram amostradas pelo comprimento da coluna que possui as alturas das crianças `length(galton$alt_crianca)`, ou então pelo número de linhas do conjunto de dados `nrow(galton)`. Dado isso, podemos executar o laço:

```
n_linhas<-1:nrow(galton)

n_linhas

for (i in n_linhas){
    altura.crianca<-galton$alt_crianca[i]
    altura.pai<-galton$alt_pai[i]
    dif<-altura.pai-altura.crianca
    print(paste0("A diferença de altura é ",dif," para a criança na posição ",i))
}
```

O vetor `n_linhas` possui números que identificam as linhas do conjunto de dados, abrangendo de 1 a 898. O index `i` assumirá um dos valores do vetor `n_linhas` a cada repetição, seguindo a ordem que os valores estão armazenados no vetor. Definido o valor de `i` numa dada repetição, a altura da criança naquela linha é armazenada em `altura.crianca`, enquanto que a altura do pai na mesma linha é armazenada em `altura.pai`. A cada repetição a diferença entre a altura do pai e da criança é armazenada no objeto `dif`, e o resultado é impresso na tela em forma de texto.

### Laços condicionais (*while loops*)

Os laços condicionais permitem repetir uma tarefa enquanto alguma condição lógica for satisfeita. Consequentemente, o número de repetições não é pré-determinado, podendo varias conforme algum critério lógico. Essas iterações precisam de condições expressas por operadores relacionais para funcionarem. A estrutura geral dos laços condicionais é:

```
while (*condição lógica*){
  *tarefa a ser repetida*
}
```

Enquanto a condição lógica for satisfeita, resultando em `TRUE`, a tarefa será repetida. Quando a condição lógica *não* for satisfeita pela primeira vez, resultando em `FALSE`, a tarefa será interrompida. Isso implica que a condição lógica deve em algum momento resultar em `FALSE`, se não a tarefa será repetida infinitamente. Caso isso ocorra, o console do R ficará indisponível para qualquer outro comando e possivelmente "quebrará" por falta de memória. Quando preso em um laço com repetições infinitas, encerre o console com a tecla "Esc" do seu teclado.

Por exemplo, se quisermos saber a diferença de altura entre cada criança e seu pai até que a primeira criança abaixo de 1,65 seja amostrada, podemos utilizar um laço condicional. Para isso, vamos calcular a diferença de altura entre cada criança e seu pai, parando essa tarefa quando acharmos a primeira criança "baixinha". Nós não sabemos quantas vezes teremos que repetir esse cálculo até achar o primeiro "baixinho". Assim, podemos executar o seguinte código:

```
linha<-1

while (galton$alt_crianca[linha] > 1.65){
    altura.crianca<-galton$alt_crianca[linha]
    altura.pai<-galton$alt_pai[linha]
    dif<-altura.pai-altura.crianca
    print(paste0("A diferença de altura é ",dif," para a criança na posição ",linha))
    linha<-linha+1
}
```

O objeto `linha` foi criado para refletir a linha que será amostrada no conjunto de dados, tendo o valor inicial 1 (um) para indicar a primeira linha. A cada repetição será avaliada se a criança amostrada é maior que 1.65 metros `while (galton$alt_crianca[linha] > 1.65)`. Se a criança for maior que 1.65 (condição satisfeita), a altura da criança é armazenado em `altura.crianca`, enquanto que a altura do seu pai é armazenada em `altura.pai`. A diferença de alturas entre pai e criança é armazenada em `dif`, e o resultado é impresso na tela em forma de texto. Por fim, o valor do objeto `linha` é atualizado com a adição de mais um `+1`, ou seja, próxima linha. Isto fará com que a próxima repetição avalie a próxima linha de dados. O processo de avaliação, cálculo, e atualização cessa apenas quando a primeira criança menor de 1.65 metros é achada.

### Controles de fluxo

Controles de fluxo determinam se uma tarefa será executada ou não. Para isso, os controles necessitam de condições lógicas. Existem dois principais operadores para implementar controles de fluxo, `if` e `else` ("se" e "senão", respectivamente). Um controle de fluxo **sempre** requer uma condição expressa por `if`, mas não necessariamente requer `else`. A estrutura geral dos controles de fluxo é:

```
if(condição lógica){
  *tarefa executada quando a condição é satisfeita*
  }else{
  *tarefa executada quando a condição não é satisfeita*
  }
```

Se a condição lógica for satisfeita, gerando o valor `TRUE`, a tarefa dentro do primeiro grupo de chaves `{}` será executada. Caso contrário, a tarefa dentro do segundo grupo de chaves `{}` será executada.

Por exemplo, se quisermos criar um esquema que indique se uma criança é baixinha ou não, podemos usar controladores de fluxo. Assumindo alturas abaixo de 1.65 como "baixinha", podemos sinalizar quando uma criança é baixinha ou não pelo seguinte código:

```
crianca<-galton[1,]

if(crianca$alt_crianca > 1.65){
    cat("A criança não é baixinha")
    }else{
    cat("A criança é baixinha")
}
```

O vetor `crianca` foi criado para armazenar os dados da criança que será avaliada. Nós podemos decidir qual criança será avaliada mudando o número dentro dos colchetes em `galton[,]`. Uma vez escolhida a criança, sua altura é usada na expressão lógica `crianca$alt_crianca > 1.65`. **Se** a altura for maior que 1.65, o resultado é `TRUE`, e a primeira tarefa é executada `print("A criança não é baixinha")`; **senão** a segunda tarefa é executada `print("A criança é baixinha")`.

### Programando por aninhamento

Os laços e os controles de fluxo são a base da programação em R. Uma forma simples de programar é aninhando esses tipos de código, ou seja, "botando um dentro do outro". Não existem limites quanto ao número de laços ou de controles de fluxo que você pode implementar. Você dispensará um bom tempo tentando achar a causa dos erros em seu programa (Acreditem, é bem normal!). Contudo, uma vez que você sanar todos os erros, você poupará muito tempo no futuro ao automatizar várias tarefas. Aqui vamos explorar um exemplo de programação por aninhamento.

Digamos que queremos classificar todas as crianças em "baixinha" ou "não baixinha", assumindo alturas abaixo de 1.65 como limiar. Nós já temos o código base para fazer essa classificação, mas o código necessita que eu escolha manualmente as crianças. Portanto, precisamos automatizar essa escolha para obter uma classificação de todas as crianças de forma rápida. Para isso, podemos executar o seguinte código:

```
classificacao<-rep(NA, length(galton$alt_crianca))

n_linhas<-1:nrow(galton)

for (i in n_linhas){
```

```
    crianca<-galton[i,]
    if(crianca$alt_crianca > 1.65){
        classificacao[i]<-c("não é baixinha")
        }else{
        classificacao[i]<-c("baixinha")
        }
    }

classificacao
```

O vetor `classificacao` é criado para armazenar a classificação das alturas das crianças. Inicialmente esse vetor não possui dados, apenas o indicador de dados faltantes `NA` (NA = *Not Available*, ou seja, não disponível). Primeiramente, um laço determinado `for` é iniciado, onde o index `i` assumirá o número das linhas do conjunto de dados. A cada repetição, o index define qual criança é escolhida, e os dados dessa criança é armazenado no vetor `crianca`. Em seguida, a altura da criança é avaliada numa condição lógica `crianca$alt_crianca > 1.65`. Se a altura da criança satisfaz a condição lógica `TRUE`, o valor "não é baixinha" é armazenado no vetor de classificação na posição relativa aquela criança. Se a altura não satisfaz a condição lógica `FALSE`, o valor "baixinha" é armazenado no vetor de classificação. Ao fim do processo, podemos chamar o vetor `classificacao` para verificar a classificação de todas as crianças.

### Funções

Por fim, após ter estabelecido um código que automatiza tarefas do nosso interesse, nós podemos querer implementar esse código numa função. As funções permitem que o código seja executado repetidas vezes sem a necessidade de escrevê-lo novamente. A implementaçaõ de funções segue o seguinte esquema:

```
nome.da.função <- funtion(argumento.1, argumento.2, ...){
  *código que será executado*
  return(*resultado final de interesse*)
}
```

As funções implementadas por nós devem ser atribuídas a um nome `nome.da.função`. A expressão `function()` indica que vamos definir uma nova função. Dentro dos parenteses `()`, nós podemos definir o nome dos argumentos que a função utilizará. Uma função pode utilizar um ou muitos argumentos. Entre as chaves `{}`, nós inlcuímos o código com as tarefas que a função irá executar, ou seja, o **corpo da função**. Os objetos que são criados no corpo da função **não** afetam os demais objetos no seu console do R, ou seja, você pode ter objetos de mesmo nome no console e no corpo da função. Isto ocorre porque os objetos do corpo da função "residem" em outro espaço da memória do R. Por fim, dentro da expressão `return()`, nós definimos o objeto que será entrege como resultado final da função e que poderá ser armazenado em um objeto do console.

Por exemplo, eu quero implementar uma função que classifique as crianças amostradas por Galton em "baixinha" ou "não baixinha", mas de acordo com um limite de altura variável. Ou seja, eu quero poder repetir a tarefa de classificação usando diferentes limites de altura. O código base para a classificação segundo a altura foi desenvolvido anteriormente (secção "Programando por aninhamento"), mas ele usa um limite fixo de altura para classificação. Assim, toda vez que eu quisesse classificar as crianças segundo um novo limite de altura, eu precisaria alterar o código base. Contudo, eu posso evitar isso ao implementar o código de classificação em uma função:

```
classifica.crianca<-function(limite){
    classificacao<-rep(NA, length(galton$alt_crianca))
    for (i in 1:length(galton$alt_crianca)){
        crianca<-galton[i,]
```

```
    if(crianca$alt_crianca > limite){
        classificacao[i]<-c("não é baixinha")
        }else{
        classificacao[i]<-c("baixinha")
        }
    }
return(classificacao)
}
```

Neste código, eu nomiei a nova função como `classifica.crianca`, uma sugestão da sua funcionalidade. Dentro dos parêntese `()` eu determinei que a função aceitaria apenas um argumento, `limite`, que especifica a altura usada como limite para classificar as crianças em "baixinha" ou "não é baixinha". Dentro do par mais inclusivo de chaves `{}` está o corpo da função, que é o código elaborado anteriormente para classificar as crianças. Contudo, perceba que dessa vez a condição lógica que avalia a altura das crianças não tem como referência um número fixo, mas sim o argumento `limite` definido no cabeçalho da função. Por fim, dentro da expressão `return()`, eu defini que o produto final da função é o vetor de classificação das crianças `classificacao`.

Para verificar se minha recém-implementada função está funcionando, eu executei a mesma com diferentes valores para o argumento `limite`.

```
classifica.crianca(limite= 1.5)

classifica.crianca(limite= 1.7)

classifica.crianca(limite= 1.8)
```

**Prática**

Chegou a hora de você desenvolver um programa no R, utilizando o conjunto de dados `Galton_dados_pt.csv`. O objetivo final do programa é gerar uma classificação das famílias que indique se a altura média das crianças é menor que a altura dos seus parentais ou não. Ou seja, a altura média dos irmãos é menor que a altura do pai *&* da mãe? O produto final do programa deve ser o vetor com a classificação das famílias. O nome das categorias dessa classificação ficam a seu critério. **DICA**: Para saber quais são as diferentes categorias/valores de um vetor, utilize a função `unique()`.