Treinamento em Programação no Ambiente R

Esse material foi elaborado para o **Treinamento em Programação no Ambiente R** organizado pelo GENt, grupo de Divulgação Científica do Programa de Pós-Graduação em Genética e Melhoramento de Plantas da ESALQ- USP.

Você pode acessar mais informações sobre o grupo neste link. No site você pode encontrar as atividades que temos desenvolvido.

Este treinamento também recebe apoio do grupo GVENCK e do Departamento de Genética da ESALQ.

Sugerimos que, antes de iniciar a prática aqui descrita, siga este tutorial para instalação do R e do RStudio.

Familiarização com a interface do RStudio

Abrindo o RStudio você verá:

A interface é separada em quatro janelas com principais funções:

- Edição de código
- Ambiente de trabalho e histórico
- Console
- Arquivos, gráficos, pacotes e ajuda

Explore cada uma das janelas. São inúmeras funcionalidades para cada uma delas, veremos algumas delas ao decorrer do curso.

Um primeiro script

A janela de edição de código (provavelmente localizada no canto superior esquerdo) você irá utilizar para escrever o seu código. Abra um novo script clicando no `+` no canto superior esquerdo e selecionando `R script`.

Vamos então iniciar os trabalhos com o tradicional `Hello World`. Digite no seu script:

```
cat("Hello world")
```

```
## Hello world
```

Agora, selecione a linha e aperte o botão `Run` ou utilize `Ctrl + enter`.

Ao fazer isso o seu código será processado na janela `Console`, onde aparecerá em azul (se você estiver com as cores padrão do R) o código escrito e, logo em seguida, o resultado desejado. A linha somente não será processada no console se houver o símbolo `#` na frente. Agora, experimente colocar `#` na frente do código escrito. E, novamente, selecione a linha e aperte `Run`.

```
# cat("Hello world")
```

O símbolo `#` é muito utilizado para realizar **comentários** ao decorrer do código. Esta é uma ótima prática para deixar o código organizado e para que você possa lembrar mais tarde o que você mesmo/a estava pensando quando o escreveu ou para que outras pessoas possam entendê-lo. Como no exemplo:

```
# Iniciando os trabalhos no R
cat("Hello world")
```

```
## Hello world
```

Importante: sempre que quiser realizar alguma alteração, edite o seu script e não diretamente no console, pois tudo o que neste é escrito, não terá como ser salvo!

Para salvar seu script, você pode utilizar a aba `Files` localizada (como padrão) no canto direito inferior para procurar uma localização de sua preferência, criar uma nova pasta com o nome `CursoR`.

Dica:

- Evite colocar espaços e pontuações no nome das pastas e arquivos, isso pode dificultar o acesso via linha de comando no R. Por exemplo, ao invés de `Curso R`, optamos por `CursoR`.

Depois, basta clicar no disquete localizado no cabeçalho do RStudio ou com `Ctrl + s` e selecionar o diretório `CursoR` criado. Scripts em R são salvos com a extensão `.R`.

Estabelecendo diretório de trabalho

Outra boa prática no R é deixar o script no mesmo diretório onde estão seus dados brutos (arquivos de entrada no script) e os dados processados (gráficos, tabelas, etc). Para isso, vamos fazer com que o R identifique o mesmo diretório em que você salvou o script como **diretório de trabalho**, assim ele entenderá que é dali que precisa obter os dados e para lá que também irão os resultados.

Você pode fazer isso utilizando as facilidades do RStudio, basta localizar o diretório `CursoR` pela aba `Files`, clique em `More` e depois "Set as Working Directory". Repare que irá aparecer no console algo como:

```
setwd("~/Documents/CursoR")
```

Ou seja, você pode utilizar este mesmo comando para realizar esta ação. Esta então será nossa pasta de trabalho. Quando estiver perdido/a ou para ter certeza que o diretório de trabalho foi alterado utilize:

```
getwd()
```

Facilitando a vida com `Tab`

Agora, imagine que você tem um diretório como `~/Documentos/mestrado/semestre1/disciplina_tal/aula_tal/dados_281`. Não é fácil lembrar todo este caminho para escrever num comando `setwd()`.

Além da facilidade da janela do RStudio, você pode utilizar a tecla `Tab` para completar o caminho para você. Experimente buscando alguma pasta no seu computador. Basta começar a digitar o caminho e apertar `Tab`, ele irá completar o nome para você! Se você tiver mais do que um arquivo com aquele início de nome, aperte duas vezes o `Tab`, ele mostrará todas as opções.

O `Tab` funciona não só para indicar caminhos, mas também para comandos e nomes de objetos. É muito comum errarmos no código por erros de digitação, utilizar o `Tab` fará com que reduza significativamente esses erros.

Operações básicas

Vamos então à linguagem!

O R pode funcionar como uma simples **calculadora**, que utiliza a mesma sintaxe que outros programas (como o excel):

```
#########################
# Script Treinamento R

# Data: 16/05/2019
# GENt
#########################
```

```r
1+1.3                  #Decimal definido com "."
2*3
2^3
4/2

sqrt(4)                #raíz quadrada
log(100, base = 10)    #logarítmo na base 10
log(100)               #logarítmo com base neperiana
```

Agora, utilize as operações básicas para solucionar expressão abaixo. Lembre-se de utilizar parênteses `()` para estabelecer prioridades nas operações.

$(\frac{13+2+1.5}{3}) + log_4 96$

Resultado esperado:

```
## [1] 8.792481
```

Os comandos `log` e `sqrt` são duas de muitas outras funções básicas que o R possui. Para todas elas o R possui uma descrição para auxiliar no seu uso, para acessar essa ajuda use:

```r
?log
```

E será aberta a descrição da função na janela `Help` do RStudio.

Se a descrição do próprio R não for suficiente para você entender como funciona a função, busque no google (de preferência em inglês). Existem diversos sites e fóruns com informações didáticas das funções do R.

Operações com vetores

Os vetores são as estruturas mais simples trabalhadas no R. Construímos um vetor com uma sequencia numérica usando:

```r
c(1,3,2,5,2)
```

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

MUITA ATENÇÃO: O *c* é a função do R (*Combine Values into a Vector or List*) com a qual construímos um vetor!

Utilizamos o simbolo `:` para criar sequencias de números inteiros, como:

```r
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Podemos utilizar outras funções para gerar sequencias, como:

```r
seq(from=0, to=100, by=5)
```

```
##  [1]   0   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80
## [18]  85  90  95 100
```

```r
# ou
seq(0,100,5) # Se você já souber a ordem dos argumentos da função
```

```
##  [1]   0   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80
## [18]  85  90  95 100
```

- Crie uma sequencia utilizando a função `seq` que varie de 4 a 30, com intervalos de 3 em 3.

```
## [1]  4  7 10 13 16 19 22 25 28
```

A função `rep` gera sequencias com números repetidos:

```
rep(3:5, 2)
```

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

Podemos realizar operações utilizando esses vetores:

```
c(1,4,3,2)*2
c(4,2,1,5)+c(5,2,6,1)
c(4,2,1,5)*c(5,2,6,1)
```

Repare que já esta ficando cansativo digitar os mesmos números repetidamente, vamos resolver isso criando **objetos** para armazenar nossos vetores e muito mais.

Criando objetos

O armazenamento de informações em objetos e a possível manipulação desses faz do R uma linguagem *orientada por objetos*. Para criar um objeto basta atribuir valores para as variáveis, como a seguir:

```
x = c(30.1,30.4,40,30.2,30.6,40.1)
# ou
x <- c(30.1,30.4,40,30.2,30.6,40.1)

y = c(0.26,0.3,0.36,0.24,0.27,0.35)
```

Os mais antigos costumam usar o sinal `<-`, mas tem a mesma função de `=`. Escolha usar o qual preferir.

Para acessar os valores dentro do objeto basta:

```
x
```

```
## [1] 30.1 30.4 40.0 30.2 30.6 40.1
```

A linguagem é sensível à letras maiúsculas e minúsculas, portanto `x` é diferente de `X`:

```
X
```

O objeto `X` não foi criado.

Podemos então realizar as operações com o objeto criado:

```
x*2
x + y
x*y
```

E podemos armazenar a operação em outro objeto:

```
z <- (x+y)/2
z
```

Podemos também aplicar algumas funções, como exemplo:

```
sum(z)  # soma dos valores de z
```

```
## [1] 101.59
```

```
mean(z) # média
```

```
## [1] 16.93167
```

```
var(z)  # variância
```

```
## [1] 6.427507
```

Acessamos somente o 3º valor do vetor criado com `[]`:

```
z[3]
```

Também podemos acessar o número da posição 2 a 4 com:

```
z[2:4]
```

```
## [1] 15.35 20.18 15.22
```

Para obter informações do vetor criado utilize:

```
str(z)
```

```
##  num [1:6] 15.2 15.3 20.2 15.2 15.4 ...
```

A função `str` nos diz sobre a estrutura do vetor, que se trata de um vetor **numérico** com 6 elementos.

Os vetores também podem receber outras categorias como **caracteres**:

```
clone <- c("GRA02", "URO01", "URO03", "GRA02", "GRA01", "URO01")
```

Outra classe são os **fatores**, esses podem ser um pouco complexos de lidar.

De forma geral, fatores são valores categorizados por `levels`, como exemplo, se transformarmos nosso vetor de caracteres `clone` em fator, serão atribuidos níveis para cada uma das palavras:

```
clone_fator <- as.factor(clone)
str(clone_fator)
```

```
##  Factor w/ 4 levels "GRA01","GRA02",..: 2 3 4 2 1 3
```

```
levels(clone_fator)
```

```
## [1] "GRA01" "GRA02" "URO01" "URO03"
```

Dessa forma, teremos apenas 4 níveis para um vetor com 6 elementos, já que as palavras "GRA02" e "URO01" se repetem. Podemos obter o número de elementos do vetor ou o seu comprimento com:

```
length(clone_fator)
```

```
## [1] 6
```

Também há vetores **lógicos**, que recebem valores de verdadeiro ou falso:

```
logico <- x > 40
logico   # Os elementos são maiores que 40?
```

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

Com ele podemos, por exemplo, identificar quais são as posições dos elementos maiores que 40:

```
which(logico)  # Obtendo as posiçoes dos elementos TRUE
```

```
## [1] 6
```

```
x[which(logico)] # Obtendo os números maiores que 40 do vetor x pela posição
```

```
## [1] 40.1
```

Encontre mais sobre outros operadores lógicos, como o `>` utilizado, neste link.

Warning1

Faça uma sequência numérica, contendo 10 valores inteiros, e salve em um objeto chamado "a".

```r
(a <- 1:10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Crie outra sequência, utilizando números decimais e qualquer operação matemática, de tal forma que seus valores sejam idênticos ao objeto "a".

```r
b <- seq(from = 0.1, to = 1, 0.1)
(b <- b*10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Os dois vetores parecem iguais, não?

Então, utilizando um operador lógico, vamos verificar o objeto "b" é igual ao objeto "a".

```r
a==b
```

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

Alguns valores não são iguais. Como isso é possivel?

```r
a==round(b)
```

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

Warning2

Não é possível misturar diferentes classes dentro de um mesmo vetor, ao tentar fazer isso repare que o R irá tentar igualar para uma única classe:

```r
errado <- c(TRUE, "vish", 1)
errado
```

```
## [1] "TRUE" "vish" "1"
```

No caso, todos os elementos foram transformados em caracter.

Algumas Dicas:

- Cuidado com a prioridade das operações, na dúvida, sempre acrescente parenteses conforme seu interesse de prioridade.
- Lembre-se que, se esquecer de fechar algum `(` ou `[` ou `"`, o console do R ficará esperando você fechar indicando um `+`, nada será processado até que você digite diretamente no console um `)`.
- Cuidado para não sobrepor objetos já criados criando outros com o mesmo nome. Use, por exemplo: altura1, altura2.
- Mantenha no seu script .R somente os comandos que funcionaram e, de preferência, adicione comentários. Você pode, por exemplo, comentar dificuldades encontradas, para que você não cometa os mesmos erros mais tarde.

  Você já pode fazer os exercícios da **Sessão 1**

Paramos aqui no primeiro dia.

- Caso não tenha salvado os objetos criados até agora, obtenha-os aqui.
- Aqui você pode acessar um exemplo de script `.R` para esse primeiro dia.
- E aqui um exemplo de arquivo `.Rmd` para gerar um relatório com o conteúdo desse primeiro dia. E suas versões compiladas em pdf e html.

Matrizes

As matrizes são outra classe de objetos muito utilizadas no R, com elas podemos realizar operações de maior escala de forma automatizada.

Por serem usadas em operações, normalmente armazenamos nelas elementos numéricos. Para criar uma matriz, determinamos uma sequência de números e indicamos o número de linhas e colunas da matriz:

```
X <- matrix(1:12, nrow = 6, ncol = 2)
X
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
## [4,]    4   10
## [5,]    5   11
## [6,]    6   12
```

Podemos também utilizar sequencias já armazenadas em vetores para gerar uma matriz, desde que eles sejam numéricos:

```
W <- matrix(c(x,y), nrow = 6, ncol =2)
W
```

```
##      [,1] [,2]
## [1,] 30.1 0.26
## [2,] 30.4 0.30
## [3,] 40.0 0.36
## [4,] 30.2 0.24
## [5,] 30.6 0.27
## [6,] 40.1 0.35
```

Com elas podemos realizar operações matriciais:

```
X*2
```

```
##      [,1] [,2]
## [1,]    2   14
## [2,]    4   16
## [3,]    6   18
## [4,]    8   20
## [5,]   10   22
## [6,]   12   24
```

```
X*X
```

```
##      [,1] [,2]
## [1,]    1   49
## [2,]    4   64
## [3,]    9   81
## [4,]   16  100
## [5,]   25  121
## [6,]   36  144
```

```
X%*%t(X)          # Multiplicação matricial
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   50   58   66   74   82   90
## [2,]   58   68   78   88   98  108
```

```
## [3,]   66   78   90  102  114  126
## [4,]   74   88  102  116  130  144
## [5,]   82   98  114  130  146  162
## [6,]   90  108  126  144  162  180
```

Utilizar essas operações exige conhecimento de álgebra de matrizes, se quiser se aprofundar a respeito, o livro *Linear Models in Statistics, Rencher (2008)* possui um boa revisão à respeito. Você também pode explorar a sintaxe do R para essas operações neste link.

Acessamos os números internos à matriz dando as coordenadas [linha,coluna], como no exemplo:

```
W[4,2] # Número posicionado na linha 4 e coluna 2
```

```
## [1] 0.24
```

As vezes pode ser informativo dar nomes às colunas e às linhas da matriz, fazemos isso com:

```
colnames(W) <- c("altura", "diametro")
rownames(W) <- clone
W
```

```
##       altura diametro
## GRA02   30.1     0.26
## URO01   30.4     0.30
## URO03   40.0     0.36
## GRA02   30.2     0.24
## GRA01   30.6     0.27
## URO01   40.1     0.35
```

Essas funções `colnames` e `rownames` também funcionam nos data.frames.

Data.frames

Diferente das matrizes, não realizamos operações com os data.frames, mas eles permitem a união de vetores com classes diferentes. Os *data frames* são semelhantes a tabelas geradas em outros programas, como o excel.

Os *data frames* são combinações de vetores de mesmo comprimento. Todos os que criamos até agora tem tamanho 6, verifique.

Podemos assim combiná-los em colunas de um único data.frame:

```
campo1 <- data.frame("clone" = clone,     # Antes do sinal de "="
                     "altura" = x,        # estabelecemos os nomes
                     "diametro" = y,      # das colunas
                     "idade" = rep(3:5, 2),
                     "corte"= logico)
campo1
```

```
##   clone altura diametro idade corte
## 1 GRA02   30.1     0.26     3 FALSE
## 2 URO01   30.4     0.30     4 FALSE
## 3 URO03   40.0     0.36     5 FALSE
## 4 GRA02   30.2     0.24     3 FALSE
## 5 GRA01   30.6     0.27     4 FALSE
## 6 URO01   40.1     0.35     5  TRUE
```

Podemos acessar cada uma das colunas com:

```
campo1$idade
```

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

Ou também com:

```
campo1[,4]
```

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

Aqui, o número dentro dos colchetes se refere à coluna, por ser o segundo elemento (separado por vírgula). O primeiro elemento se refere à linha. Como deixamos o primeiro elemento vazio, estaremos nos referindo a todas as linhas para aquela coluna.

Dessa forma, se quisermos obter um conteúdo específico podemos dar as coordenadas com [linha,coluna]:

```
campo1[1,2]
```

```
## [1] 30.1
```

- Obtenha o diâmetro do clone "URO03.

```
## [1] 0.36
```

Mesmo se tratando de um *data frame*, podemos realizar operações com os vetores numéricos que a compõe.

- Com o diâmetro e a altura das árvores, calcule o volume conforme a fórmula a seguir e armazene em um objeto `volume`:

$3.14 * (diametro/2)^2 * altura$

```
## [1] 1.597287 2.147760 4.069440 1.365523 1.751131 3.856116
```

Agora, vamos adicionar o vetor calculado com o volume ao nosso data.frame. Para isso use a função `cbind`.

```
campo1 <- cbind(campo1, volume)
str(campo1)
```

```
## 'data.frame':    6 obs. of  6 variables:
##  $ clone   : Factor w/ 4 levels "GRA01","GRA02",..: 2 3 4 2 1 3
##  $ altura  : num  30.1 30.4 40 30.2 30.6 40.1
##  $ diametro: num  0.26 0.3 0.36 0.24 0.27 0.35
##  $ idade   : int  3 4 5 3 4 5
##  $ corte   : logi  FALSE FALSE FALSE FALSE FALSE TRUE
##  $ volume  : num  1.6 2.15 4.07 1.37 1.75 ...
```

Algumas dicas:

- Lembre-se que, para construir matrizes e *data frames*, o número de elementos em cada coluna tem que ser iguais.
- Caso não saiba o operador ou a função que deve ser utilizada, como o desvio padrão, busque no google algo como "desvio padrão R", ou melhor "standard deviation R". Logo nas primeiras páginas você obterá respostas. A comunidade do R é bastante ativa e grande parte das suas perguntas sobre ele já foram respondidas em algum lugar da web.
- Não esqueça que tudo o que fizer no R precisa ser explicitamente indicado, como uma multiplicação 4ac com `4*a*c`. Para gerar um vetor 1,3,2,6 é necessário: `c(1,3,2,6)`.

Listas

Listas consistem em uma coleção de objetos, não necessariamente de mesma classe. Nelas podemos armazenar todos os outros objetos ja vistos e recuperá-los pela indexação com `[[`. Como exemplo, vamos utilizar alguns

objetos que já foram gerados.

```
minha_lista <- list(campo1 = campo1, media_alt = tapply(campo1$altura, campo1$idade, mean), matrix_ex =
str(minha_lista)
```

```
## List of 3
##  $ campo1   :'data.frame':   6 obs. of  6 variables:
##   ..$ clone   : Factor w/ 4 levels "GRA01","GRA02",..: 2 3 4 2 1 3
##   ..$ altura  : num [1:6] 30.1 30.4 40 30.2 30.6 40.1
##   ..$ diametro: num [1:6] 0.26 0.3 0.36 0.24 0.27 0.35
##   ..$ idade   : int [1:6] 3 4 5 3 4 5
##   ..$ corte   : logi [1:6] FALSE FALSE FALSE FALSE FALSE TRUE
##   ..$ volume  : num [1:6] 1.6 2.15 4.07 1.37 1.75 ...
##  $ media_alt: num [1:3(1d)] 30.1 30.5 40
##   ..- attr(*, "dimnames")=List of 1
##   .. ..$ : chr [1:3] "3" "4" "5"
##  $ matrix_ex: num [1:6, 1:2] 30.1 30.4 40 30.2 30.6 40.1 0.26 0.3 0.36 0.24 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:6] "GRA02" "URO01" "URO03" "GRA02" ...
##   .. ..$ : chr [1:2] "altura" "diametro"
```

Quero acessar o data.frame `campo1`

```
minha_lista[[1]]
```

```
##   clone altura diametro idade corte   volume
## 1 GRA02   30.1     0.26     3 FALSE 1.597287
## 2 URO01   30.4     0.30     4 FALSE 2.147760
## 3 URO03   40.0     0.36     5 FALSE 4.069440
## 4 GRA02   30.2     0.24     3 FALSE 1.365523
## 5 GRA01   30.6     0.27     4 FALSE 1.751131
## 6 URO01   40.1     0.35     5  TRUE 3.856116
```

```
# ou
minha_lista$campo1
```

```
##   clone altura diametro idade corte   volume
## 1 GRA02   30.1     0.26     3 FALSE 1.597287
## 2 URO01   30.4     0.30     4 FALSE 2.147760
## 3 URO03   40.0     0.36     5 FALSE 4.069440
## 4 GRA02   30.2     0.24     3 FALSE 1.365523
## 5 GRA01   30.6     0.27     4 FALSE 1.751131
## 6 URO01   40.1     0.35     5  TRUE 3.856116
```

Listas são muito úteis, por exemplo, quando vamos utilizar/gerar diversos objetos diferentes dentro de um loop.

Arrays

Este é um tipo de objeto que você provavelmente não irá utilizar agora no início, mas é bom saber da sua existência. São utilizados para armazenar dados com mais de duas dimensões. Por exemplo, se criarmos um array:

```
(meu_array <- array(1:24, dim = c(2,3,4)))
```

```
## , , 1
##
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

Teremos quatro matrizes com duas linhas e três colunas e os números de 1 a 24 estarão distribuídos nelas por colunas.

Agora você já pode fazer os exercícios da **Sessão 2**

Exportando e importando dados

Os objetos criados dentro do R podem ser exportados em arquivos de diversos formatos. Lembre-se que, se não definir todo o caminho que deseja depositar o arquivo, ele irá ser salvo no seu diretório de trabalho.

Para exportar o objeto no formato .RData:

```
save(campo1, file = "campo1.RData")
```

Essa é uma forma de salvar o objeto trabalhado, se removermos ele com:

```
rm(campo1)  # Certifique-se que salvou o objeto antes de removê-lo
```

Podemos facilmente obtê-lo novamente com:

```
load("campo1.RData")
```

Para salvar todos os objetos do seu *workspace* use:

```
save.image()
```

O R irá criar um arquivo `.Rdata` contendo todos os seus objetos.

Podemos exportar nossos objetos em outros formatos, como, por exemplo, `.txt` ou `.csv`. Para isso utilizamos:

```
write.table(campo1, file = "campo1.txt", sep = ";", dec = ".", row.names = FALSE)
write.csv(campo1, file = "campo1.csv", row.names = TRUE)
```

Obs: Você pode adquirir pacotes para exportar e importar dados com outros fomatos, como exemplo o pacote `xlsx` exporta e importa dados com formato do excel.

Ao exportar, há diversas opções para a formatação do arquivo, é importante considerá-las se o arquivo for ser trabalhado em outro sofware posteriormente.

Abra os arquivos gerados para visualizar sua formatação.

Esses arquivos podem ser lidos novamente pelo R, utilizando as funções e suas especificações:

```
campo1_txt <- read.table(file = "campo1.txt", sep=";", dec=".", header = TRUE)
campo1_csv <- read.csv(file = "campo1.csv")
head(campo1_txt)
head(campo1_csv)
```

Agora que aprendemos a importar dados, vamos trabalhar com o conjunto gerado a partir do formulário que vocês preencheram.

A planilha com os dados será disponibilizada, adicione-a ao seu diretório de trabalho ou indique o caminho da pasta ao importá-la para dentro do R, como a seguir. Caso esteja difícil encontrá-la, acesse:

- dados.RData
- dados.csv

Aqui usaremos o argumento `stringAsFactors` que impede que o R transforme os vetores da tabela em fatores, os quais são mais difíceis de trabalhar. O argumento `na.strings` irá indicar como foram nomeados os dados perdidos.

```
dados <- read.csv(file = "dados.csv", stringsAsFactors = FALSE, na.strings="-")
```

```
load("dados_alunos.RData")
```

Vamos explorar a estrutura dos dados coletados:

```
str(dados)
```

```
## 'data.frame':    35 obs. of  12 variables:
##  $ Timestamp                                                          : chr  "5/16/2019
##  $ Idade..Ex..26.                                                     : int  38 27 26 1
##  $ Dia.e.mês.do.aniversário..Ex..05.10.                               : chr  "02/01" "1
##  $ Gênero                                                             : chr  "Masculino
##  $ Cidade.de.Origem..Ex..Piracicaba.SP.                               : chr  "Piracicab
##  $ Altura.em.metros..Ex..1.60.                                        : num  1.82 1.5 1
##  $ Peso.em.Kg..Ex..56.                                                : int  91 58 56 5
##  $ Área.com.a.qual.mais.se.identifica                                 : chr  "Biológica
##  $ Dê.uma.nota.de.0.a.10.para.seu.nível.de.conhecimento.em.R           : int  3 2 1 2 4
##  $ Você.utiliza.alguma.outra.linguagem.de.programação..Qual.is...Ex..C..C....python.: chr  "Não" "Não
##  $ Para.que.você.utiliza.ou.utilizará.o.R.                            : chr  "programa
##  $ Qual.a.sua.motivação.para.fazer.este.curso...texto.de.30.a.100.palavras.  : chr  "material
```

```
# também
dim(dados)
```

```
## [1] 35 12
```

Repare que nos nomes das colunas ainda estão as perguntas completas feitas no formulário, vamos alterar para nomes mais fáceis de trabalhar:

```
colnames(dados) <- c("Data_pesq", "Idade", "Niver", "Genero", "Cidade",
                     "Altura","Peso", "Area", "ConhecimentoR", "Outras_linguagens",
                     "Utilizacao", "Motivacao")
str(dados)
```

```
## 'data.frame':    35 obs. of  12 variables:
```

```
##  $ Data_pesq        : chr  "5/16/2019 11:19:37" "5/16/2019 11:30:40" "5/16/2019 14:21:20" "5/17/2019
##  $ Idade            : int  38 27 26 24 25 34 45 22 31 21 ...
##  $ Niver            : chr  "02/01" "28/05" "21/04" "07/02" ...
##  $ Genero           : chr  "Masculino" "Feminino" "Feminino" "Feminino" ...
##  $ Cidade           : chr  "Piracicaba-SP" "Guaxupé - MG" "São José dos Campos" "Alta Floresta - MT"
##  $ Altura           : num  1.82 1.5 1.56 1.64 1.7 1.64 1.88 1.81 1.73 1.63 ...
##  $ Peso             : int  91 58 56 58 54 56 93 85 75 58 ...
##  $ Area             : chr  "Biológicas" "Biológicas" "Interdiciplinar" "Interdiciplinar" ...
##  $ ConhecimentoR    : int  3 2 1 2 4 2 0 2 3 1 ...
##  $ Outras_linguagens: chr  "Não" "Não" "Não" "Não" ...
##  $ Utilizacao       : chr  "programa livre" "Tese de Doutorado" "Análise de dados de pesquisa" "Anál
##  $ Motivacao        : chr  "material mais robusto para análise de dados" "Importância de aprendizado
```

Paradoxo do aniversário

Nossa primeira análise com esses dados envolverá um problema denominado "Paradoxo do aniversário", que afirma que em um grupo de 23 pessoas (ou mais), escolhidas aleatoriamente, há mais de 50% de chance de duas pessoas terem a mesma data de aniversário.

Primeiro, vamos verificar em quantos somos, contando o número de linhas, para isso use a função `nrow`.

```
nrow(dados)
```

```
## [1] 35
```

Vamos então verificar se temos no nosso grupo pessoas que compartilham o mesmo dia de aniversário.

Podemos verificar isso facilmente com a função `table`, que indica a frequência de cada observação:

```
table(dados$Niver)
```

Estruturas condicionais

if e else

Para nossa próxima atividade com os dados, vamos primeiro entender como funcionam as estruturas `if` e `else`.

Nas funções condicionais `if` e `else`, estabelecemos uma condição para if, se ela for verdade a atividade será realizada, caso contrário (else) outra tarefa será. Como no exemplo:

```
if(2 >3){
  print("dois é maior que três")
} else {
  print("dois não é maior que três")
}
```

```
## [1] "dois não é maior que três"
```

- Teste o nível de conhecimento em R obtidos no formulaŕio (9ª coluna) pela terceira pessoa que o respondeu (linha 3). Envie uma mensagem motivacional se ela não possuir qualquer conhecimento (nota 0), outra se possuir algum conhecimento (restante das notas). (dica: o sinal `==` se refere a "exatamente igual a")

```r
if(dados[3,9] == 0){
  print("Nunca é tarde para começar!")
} else {
  print("Já pegou o embalo, agora é só continuar!")
}
```

```
## [1] "Já pegou o embalo, agora é só continuar!"
```

Podemos especificar mais do que uma condição repetindo a estrutura `if else`:

```r
if(dados[7,9] == 0){
  print("Nunca é tarde para começar!")
} else if (dados[7,9] > 0 && dados[7,9] < 5){
  print("Já pegou o embalo, agora é só continuar!")
} else {
  print("Nos avise se estivermos falando algo errado...hehe")
}
```

```
## [1] "Nunca é tarde para começar!"
```

Switch

Uma outra estrutura que também pode ser usada com o mesmo propósito é o `switch`. Esta estrutura é mais utilizada quando trabalhado com caracteres. Por isso vamos aplicá-la para explorar a área (8ª coluna) com que a quinta pessoa se identifica.

```r
switch(dados[5,8],
  Exatas = print("Será que aprendeu alguma linhagem de programação na graduação?"),
  Interdiciplinar = print("Em que foi a gradução?"),
  print("Ta aqui colocando o pezinho na exatas")
)
```

```
## [1] "Ta aqui colocando o pezinho na exatas"
```

A estrutura `switch` costuma ser mais rápida que o `if` e `else`. Quando lidamos com grande quantidade de dados isso pode ser uma grande vantagem.

Mas repare que só é possível utilizar essas estruturas para um elemento individual do vetor, se quisermos percorrer o vetor inteiro precisamos recorrer a outro recurso.

Estruturas de repetição

For

Esse recurso pode ser a função `for`, uma função muito utilizada e poderosa. Ela constitui uma estrutura de loop, pois irá aplicar a mesma atividade repetidamente até atingir uma determinada condição. Veja exemplos:

```r
for(i in 1:10){
  print(i)
}
```

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

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```
test <- vector()
for(i in 1:10){
  test[i] <- i+4
}
test
```

```
##  [1]  5  6  7  8  9 10 11 12 13 14
```

Nos casos acima, `i` funciona como um index que irá variar de 1 até 10 a operação determinada entre chaves.

Com essa estrutura, podemos repetir a operação realizada com as estruturas `if` e `else` para todo o vetor:

```
for(i in 1:nrow(dados)){
  if(dados[i,9] == 0){
    print("Nunca é tarde para começar!")
  } else if (dados[i,9] > 0 && dados[i,9] < 5){
    print("Já pegou o embalo, agora é só continuar!")
  } else {
      print("Nos avise se estivermos falando algo errado...hehe")
    }
}
```

```
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Nunca é tarde para começar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Nos avise se estivermos falando algo errado...hehe"
## [1] "Nunca é tarde para começar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Nos avise se estivermos falando algo errado...hehe"
## [1] "Nunca é tarde para começar!"
## [1] "Nos avise se estivermos falando algo errado...hehe"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Nunca é tarde para começar!"
## [1] "Já pegou o embalo, agora é só continuar!"
```

```
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Nunca é tarde para começar!"
## [1] "Nunca é tarde para começar!"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Nos avise se estivermos falando algo errado...hehe"
## [1] "Já pegou o embalo, agora é só continuar!"
## [1] "Já pegou o embalo, agora é só continuar!"
```

Dica: Identação

Repare a diferença:

```
# Sem identação
for(i in 1:nrow(dados)){
if(dados[i,9] == 0){
print("Nunca é tarde para começar!")
} else if (dados[i,9] > 0 && dados[i,9] < 5){
print("Já pegou o embalo, agora é só continuar!")
} else {
print("Nos avise se estivermos falando algo errado...hehe")
}
}

# Com identação correta
for(i in 1:nrow(dados)){
  if(dados[i,9] == 0){
    print("Nunca é tarde para começar!")
  } else if (dados[i,9] > 0 && dados[i,9] < 5){
    print("Já pegou o embalo, agora é só continuar!")
  } else {
    print("Nos avise se estivermos falando algo errado...hehe")
  }
}
```

Agora vamos trabalhar com a coluna 5, que possui a informação da cidade de origem dos participantes. Repare que alguns não colocaram o estado, como o exemplo sugeria. Vamos utilizar um loop para descobrir quais estão faltando. Vamos utilizar a função `grepl` para identificar as strings que contém o caracter "-", aqueles que tiverem consideraremos correto, os que não tiverem, vamos pedir para adicionar mais informações.

```
# Exemplo do uso da função grepl
grepl("-", dados[1,5]) # A primeira linha contem o caracter "-"
```

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

```
for(i in 1:nrow(dados)){
  if(grepl("-", dados[i,5])){
    cat("Esse/a seguiu o exemplo direitinho. Parabéns!\n")
  } else {
    cat("Precisamos adicionar mais informações na linha", i, "\n")
  }
}
```

```
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 3
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
```

```
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 7
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 10
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 13
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 15
## Precisamos adicionar mais informações na linha 16
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 20
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 24
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 27
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 30
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
```

Para que seja possível imprimir conteúdo de objetos durante o loop, usamos a função `cat`, ela não separa cada resposta em uma linha, precisamos colocar o `\n` indicando a quebra de linha.

Agora vamos nos mesmos corrigir essas informações. Podemos armazenar uma variável a posição das linhas incorretas, então corrigiremos manualmente somente essas:

```
corrigir <- vector()
for(i in 1:nrow(dados)){
  if(grepl("-", dados[i,5])){
    cat("Esse/a seguiu o exemplo direitinho. Parabéns!\n")
  } else {
    cat("Precisamos adicionar mais informações na linha", i, "\n")
    corrigir <- c(corrigir, i)
  }
}
```

```
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 3
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 7
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
```

```
## Precisamos adicionar mais informações na linha 10
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 13
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 15
## Precisamos adicionar mais informações na linha 16
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 20
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 24
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 27
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Precisamos adicionar mais informações na linha 30
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
## Esse/a seguiu o exemplo direitinho. Parabéns!
```

Paramos aqui na parte da manhã do segundo dia do treinamento.

- Caso não tenha salvado os objetos criados até agora, obtenha-os aqui.
- Aqui você pode acessar um exemplo de script `.R`.
- E aqui um exemplo de arquivo `.Rmd` para gerar um relatório com o conteúdo até agora. E suas versões compiladas em pdf e html.

  Como você faria para corrigir esses elementos errados? Tente!

Dica: Primeiro busque quais são as cidades desses elementos (dados[corrigir,5]), depois, faça um novo vetor com os nomes corretos e substitua na tabela

Uma possibilidade de resposta:

```
dados[corrigir,5]
```

```
##  [1] "São José dos Campos" "Piracicaba"          "São Paulo"
##  [4] "Piracicaba"          "Uberlandia"          "Piracicaba"
##  [7] "Uberaba"             "São Luis"            "Piracicaba"
## [10] "Piracicaba"
```

```
novo <- c("São José dos Campos - SP", "Piracicaba - SP", "São Paulo - SP", "Piracicaba-SP",
          "Uberlância-MG", "Piracicaba-SP", "Uberaba-MG", "São Luis-MA", "Piracicaba-SP",
          "Piracicaba-SP")
dados[corrigir,5] <- novo

# Verificando se corrigiu
dados[,5]
```

```
##  [1] "Piracicaba-SP"                "Guaxupé - MG"
```

```
##  [3] "São José dos Campos - SP"       "Alta Floresta - MT"
##  [5] "Goiania-Go"                     "Guaíra-SP"
##  [7] "Piracicaba - SP"                "Santos-SP"
##  [9] "Cuiabá-MT"                      "São Paulo - SP"
## [11] "Toledo-PR"                      "Sao Joao Del Rei- Minas Gerais"
## [13] "Piracicaba-SP"                  "Brasília - DF"
## [15] "Uberlância-MG"                  "Piracicaba-SP"
## [17] "Brasília - DF"                  "Jatai-GO"
## [19] "Mogi das Cruzes-SP"             "Uberaba-MG"
## [21] "Belo Jardin-PE"                 "Ji-Paraná/RO"
## [23] "Pará de Minas - MG"             "São Luis-MA"
## [25] "Afogados da Ingazeira-PE"       "Campinas-SP"
## [27] "Piracicaba-SP"                  "Goiatuba-GO"
## [29] "Americana-SP"                   "Piracicaba-SP"
## [31] "Piracicaba-SP"                  "Itajubá-MG"
## [33] "Nova Monte Verde - MT"          "São Paulo - SP"
## [35] "São Paulo - SP"
```

A coluna 2 da tabela se refere à idade dos participantes, imprima na tela (usando print ou cat) a década em que cada um nasceu, como a seguir: "Nasceu na década de 80".

Dica: Faça um novo vetor com a subtração das idades pelo ano atual e depois faça um loop com uma condicional para imprimir as mensagens na tela

Uma possibilidade de resposta:

```r
decada <- 2019 - dados$Idade

for(i in 1:length(decada)){
  if(decada[i] > 1960 && decada[i] < 1970){
    print("Nasceu na década de 60")
  } else if(decada[i] >= 1970 && decada[i] < 1980){
    print("Nasceu na década de 70")
  } else if(decada[i] >= 1980 && decada[i] < 1990){
    print("Nasceu na década de 80")
  } else if(decada[i] >= 1990 && decada[i] < 2000){
    print("Nasceu na década de 90")
  } else {
    print("Xóvem")
  }
}
```

```
## [1] "Nasceu na década de 80"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 80"
## [1] "Nasceu na década de 70"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 80"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 80"
## [1] "Nasceu na década de 80"
## [1] "Nasceu na década de 90"
```

```
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 80"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 80"
## [1] "Nasceu na década de 60"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 80"
## [1] "Nasceu na década de 80"
## [1] "Nasceu na década de 60"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
## [1] "Nasceu na década de 90"
```

While

Nesse tipo de estrutura de repetição a tarefa será realizada até que seja atingida determinada condição.

```
x <- 1

while(x < 5){
  x <- x + 1
  print(x)
}
```

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

É muito importante que nessa estrutura a condição seja atingida, caso contrário o loop irá funcionar infinitamente e você terá que interrompê-lo por meios externos, como, se este utilizando RStudio, clicar no simbolo em vermelho no canto direito superior da janela do console, ou apertar Ctrl+C no console.

Não é muito difícil disso acontecer, basta um pequeno erro como:

```
x <- 1

while(x < 5){
  x + 1
  print(x)
}
```

Aqui podemos utilizar os comandos `break` e `next` para atender a outras condições, como:

```
x <- 1

while(x < 5){
```

```
  x <- x + 1
  if(x==4) break
  print(x)
}
```

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

```
x <- 1

while(x < 5){
  x <- x + 1
  if(x==4) next
  print(x)
}
```

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

Repeat

Esta estrutura também exige uma condição de parada, mas esta condição é necessariamente colocada dentro do bloco de código com o uso do `break`. Ela então repete o bloco de código até a condição o interrompa.

```
x <- 1
repeat{
  x <- x+1
  print(x)
  if(x==4) break
}
```

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

Loops dentro de loops

É possível também utilizarmos estruturas de repetição dentro de estruturas de repetição. Por exemplo, se quisermos trabalhar tanto nas colunas como nas linhas de uma matrix.

```
# Criando uma matrix vazia
ex_mat <- matrix(nrow=10, ncol=10)

# cada número dentro da matrix será o produto no índice da coluna pelo índice da linha
for(i in 1:dim(ex_mat)[1]) {
  for(j in 1:dim(ex_mat)[2]) {
    ex_mat[i,j] = i*j
  }
}
```

Fizemos um vídeo com mais detalhes sobre loops no R, aumentem nossa quantidade de views e likes por lá.

Faça os exercícios da **Sessão 3**

Algumas dicas:

- Cuidado ao rodar o mesmo comando mais de uma vez, algumas variáveis podem não ser mais como eram antes. Para que o comando funcione da mesma forma é necessário que os objetos de entrada estejam da forma como você espera.
- Lembrem-se que = é para definir objetos e == é o sinal de igualdade.
- Nas estruturas condicionais e de repetição, lembrem-se que é necessário manter a sintaxe esperada: If(){} e for(i in 1:10){}. No *for*, podemos trocar a letra que será o índice, mas é sempre necessário fornecer uma sequência de inteiros contínua.
- Usar identação ajuda a visualizar o começo e fim de cada estrutura de código e facilita o abrir e fechar de chaves. Identação são aqueles espaços que usamos antes da linha, como:

```
# Criando uma matrix vazia
ex_mat <- matrix(nrow=10, ncol=10)

# cada número dentro da matrix será o produto no índice da coluna pelo índice da linha
for(i in 1:dim(ex_mat)[1]) {   # Primeiro nível, não tem espaço
  for(j in 1:dim(ex_mat)[2]) { # Segundo nível tem um espaço (tab)
    ex_mat[i,j] = i*j          # Terceiro nível tem dois espaços
  }                            # Fechei o segundo nível
}                              # Fechei o primeiro nível
```

Elaboração de funções

Normalmente é uma boa prática criar um bloco de código se vai realizar aquela ação poucas vezes. Se for realizar várias vezes a ação e de uma vez só, vale a pena fazer um loop. Mas, se for realizar diversas vezes e o objeto de entrada for modificado vale a pena fazer uma função. E, na hierarquia, quando tiver acumulado muitas funções para realizar uma tarefa mais complexa, vale a pena construir um pacote.

Aqui não vamos nos aprofundar muito nesse assunto, talvez ele renda um outro módulo inteiro, vamos apenas passar noções básicas.

A função também é considerada um objeto no R, portanto você a atribui a uma variável, nesse caso à `quadra`. Então estabelecemos os argumentos da função, nesse caso `x`. Entre as chaves fica todo o corpo da função. Se você quer que a função retorne algum valor, é necessário utilizar o `return`.

```
quadra <- function(x){
  z <- x*x
  return(z)
}

quadra(3)
```

```
## [1] 9
```

```
quadra(4)
```

```
## [1] 16
```

```
qualquer_nome <- 4
quadra(qualquer_nome)
```

```
## [1] 16
```

Vamos complicar um pouco e também dar mais sentido para construir uma função. Vamos definir no seu corpo várias ações antes de retornar o valor, algo que não poderia ser feito usando uma única função já pronta do R.

```r
## Calcula o índice de massa corporal (IMC) dos participantes
IMC <- dados$Peso/quadra(dados$Altura)

## Calcula a média das idade dos participantes
id_med <- mean(dados$Idade)

## Calcula a mediana das idades dos participantes
id_median <- median(dados$Idade)

## Calcula a porgentagem de mulheres entre os participantes
mul <- 100*(length(which(dados$Genero == "Feminino"))/length(dados$Genero))

## Faz uma lista com todos os resultados
final_list <- list(IMC=IMC, idade_media = id_med,
                   idade_mediana = id_median, porcentagem_mulheres = mul)
```

Para montarmos a função primeiro pensamos qual será o argumento de entrada, nesse caso, nosso arquivo de dados, um *data frame* contendo pelo menos as colunas `Peso`, `Altura`, `Idade` e `Genero`.

```r
minha_funcao <- function(df.entrada){
  ## Calcula o índice de massa corporal (IMC) dos participantes
  IMC <- df.entrada$Peso/quadra(df.entrada$Altura)

  ## Calcula a média das idade dos participantes
  id_med <- mean(df.entrada$Idade)

  ## Calcula a mediana das idades dos participantes
  id_median <- median(df.entrada$Idade)

  ## Calcula a porgentagem de mulheres entre os participantes
  mul <- 100*(length(which(df.entrada$Genero == "Feminino"))/length(df.entrada$Genero))

  ## Faz uma lista com todos os resultados
  final_list <- list(IMC=IMC, idade_media = id_med,
                     idade_mediana = id_median, porcentagem_mulheres = mul)
  return(final_list)

}

test_list <- minha_funcao(df.entrada = dados)
test_list
```

```
## $IMC
##  [1] 27.47253 25.77778 23.01118 21.56454 18.68512 20.82094 26.31281
##  [8] 25.94548 25.05931 21.82995 25.95156 26.60971 22.34352 23.19109
## [15] 21.51386 20.61313 22.20408 25.71166 28.69265 21.04805 24.38237
## [22] 25.55885 24.52435 24.21229 24.80159 21.35780 22.58955 17.54187
## [29] 21.48437 20.93664 19.78997 20.44444 21.71807 29.98359 22.03857
##
## $idade_media
## [1] 29.85714
##
## $idade_mediana
## [1] 27
```

```
##
## $porcentagem_mulheres
## [1] 60
```

Se é uma função para uso próprio, você saberá como deve ser o objeto de entrada, mas se ela for utilizada por outras pessoas, será necessário, além de uma prévia explicação de suas ações, verificar se o objeto de entrada esta de acordo com o esperado pela função.

```
minha_funcao <- function(df.entrada){

  if (length(grep("Altura", colnames(df.entrada))) == 0 ||
      length(grep("Peso", colnames(df.entrada))) == 0 ||
      length(grep("Idade", colnames(df.entrada))) == 0 ||
      length(grep("Genero", colnames(df.entrada))) == 0)
    stop("Esta faltando alguma das informações.")


  ## Calcula o índice de massa corporal (IMC) dos participantes
  IMC <- df.entrada$Peso/quadra(df.entrada$Altura)

  ## Calcula a média das idade dos participantes
  id_med <- mean(df.entrada$Idade)

  ## Calcula a mediana das idades dos participantes
  id_median <- median(df.entrada$Idade)

  ## Calcula a porgentagem de mulheres entre os participantes
  mul <- 100*(length(which(df.entrada$Genero == "Feminino"))/length(df.entrada$Genero))

  ## Faz uma lista com todos os resultados
  final_list <- list(IMC=IMC, idade_media = id_med,
                     idade_mediana = id_median, porcentagem_mulheres = mul)
  return(final_list)

}

test_list <- minha_funcao(df.entrada = dados)
```

```
dados1 <- dados[,-2] # Removendo coluna de idade

test_list <- minha_funcao(df.entrada = dados1)
```

Para saber mais sobre desenvolvimento de funções acesse aqui e, um pouco mais avançado, aqui.

Paramos aqui na parte da tarde do segundo dia do treinamento.

- Caso não tenha salvado os objetos criados até agora, obtenha-os aqui.
- Aqui você pode acessar um exemplo de script `.R`.
- E aqui um exemplo de arquivo `.Rmd` para gerar um relatório com o conteúdo até agora. E suas versões compiladas em pdf e html.

Rodando outros scripts .R

As vezes, parte do seu código demanda que você chame algo que foi rodado em outro script. Muitas pessoas também tem o costume de salvar as funções próprias em um script separado. Vamos fazer isso?

- Abra um novo script .R, copie suas funções para ele e o salve como `funcoes.R`

Agora, você pode acessa-las usando:

```
source("funcoes.R")
```

> Faça os exercícios da **Sessão 4**

Elaboração de gráficos simples

Para outros dados coletados, vamos gerar alguns gráficos simples utilizando as funções básicas do R. Existem pacotes como o `ggplot2`, `plotly` e `shiny` que possuem ferramentas muito poderosas para construção de gráficos, mas exigem um pouco mais de tempo para aprendizagem de sua sintaxe.

Os tipos mais comuns já possuem funções próprias, mas outros gráficos podem ser customizados de acordo com a necessidade do usuário. Vamos iniciar com um simples gráfico de frequências (ou histograma) para os dados de `Altura`.

```
hist(dados$Altura)
```

Vamos adicionar alguns argumentos para dar uma personalizada:

- `breaks` para definir os intervalos para cada barra;

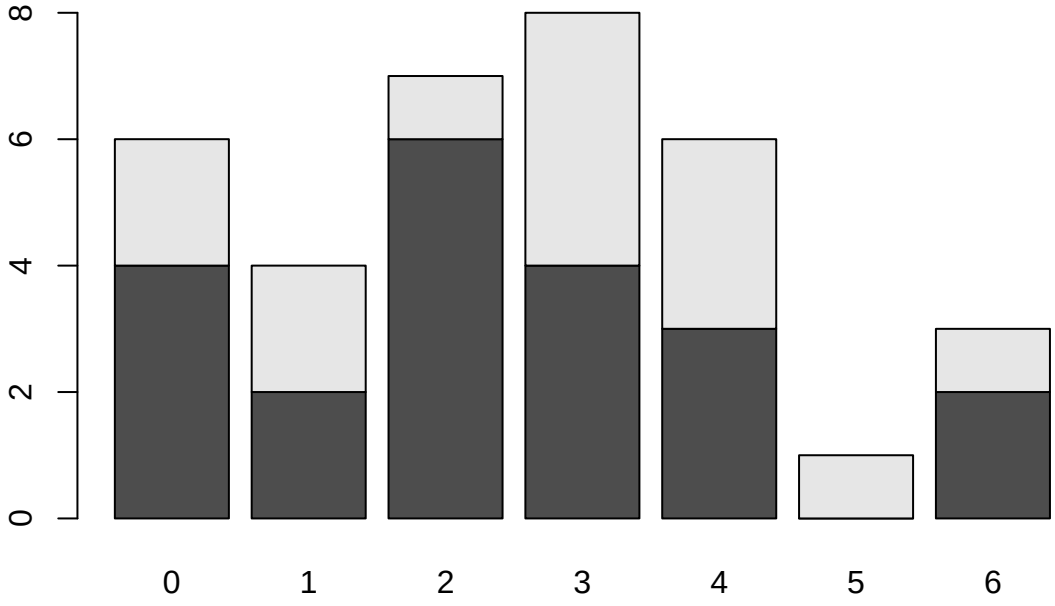
```
#### Histograma ####
#Utiliza um vetor de valores para obter as frequências
hist(dados$Altura)
hist(dados$Altura, breaks = 2)
hist(dados$Altura, breaks = 15)
```

Agora tente fazer o **histograma para o peso**, aproveite para tentar alterar alguns parâmetros. Em seguida, serão apresentados outros gráficos que poderão ser utilizados.

```
#### Bar plot ####
#É um gráfico interessante quando possuímos resultados da função table
gen <- table(dados$Genero)
barplot(gen)
```


```
#Alterando o eixo
barplot(gen, horiz = TRUE)

#Mais informações podem ser combinadas
nota_gen <- table(dados$ConhecimentoR, dados$Genero)
barplot(nota_gen)
```

Busque uma maneira de colocar a nota do R no eixo x e o gênero nas cores.




```
#### Pizza!!! ####
#Requer um vetor de contagens e os rótulos associados
pie(gen, names(gen))

#Tente criar o gráfico de pizza com a nota do R
```

```
#### Dot plots ####
#Utilizaremos um vetor de valores e um vetor de rótulos
```

```r
dotchart(dados$Altura, labels = dados$Cidade)
#Também é possível criar grupos
dotchart(dados$Altura, labels = dados$Cidade, groups = as.factor(dados$Genero))

#Podemos criar uma coluna no data frame onde delimitaremos uma cor para cada uma das áreas:
dados$color[dados$Area=="Exatas"] <- "darkblue"
dados$color[dados$Area=="Biologicas"] <- "darkgreen"
dados$color[dados$Area=="Humanas"] <- "orange"
dados$color[dados$Area=="Interdiciplinar"] <- "red"
dados$color[dados$Area=="Agrarias"] <- "black"
dotchart(dados$Altura, labels = dados$Cidade, groups = as.factor(dados$Area), color = dados$color)

#Crie um objeto que corresponda ao data.frame 'dados' ordenado de acordo com a Altura (ou peso, como de
#Em seguida, tente criar o dot plot.

#### Curve ####
#Utiliza uma expressão/função
curve(x^2 + 3*x)
curve(2*x, add = TRUE, col="red")

#### Boxplot ####
#Necessita de um vetor de valores ou uma expressão
boxplot(dados$Altura)
boxplot(dados$Altura~dados$Genero)

#Explore o bloxplot de outros atributos, podendo combiná-los
#Desafio: faça o boxplot da altura pela combinação dos grupos Genero e Area
boxplot(Altura~Genero*Area, data = dados)

#### Scatter plot ####
#Utilizaremos coordenadas de pontos.
plot(x = dados$Peso, y= dados$Altura)
```

Parâmetros gráficos

É possível customizar os gráficos alterando seus eixos, cores, fontes, etc. Começaremos através do histograma:

- `col` para adicionar cor;
- `xlab` alterar título do eixo x;
- `ylab` alterar título do eixo y;
- `main` alterar título geral;

```r
hist(dados$Idade, breaks=6, col="blue", xlab="Idades",
     ylab="Frequencia", main="Histograma das Idades (Curso 2019)")
```

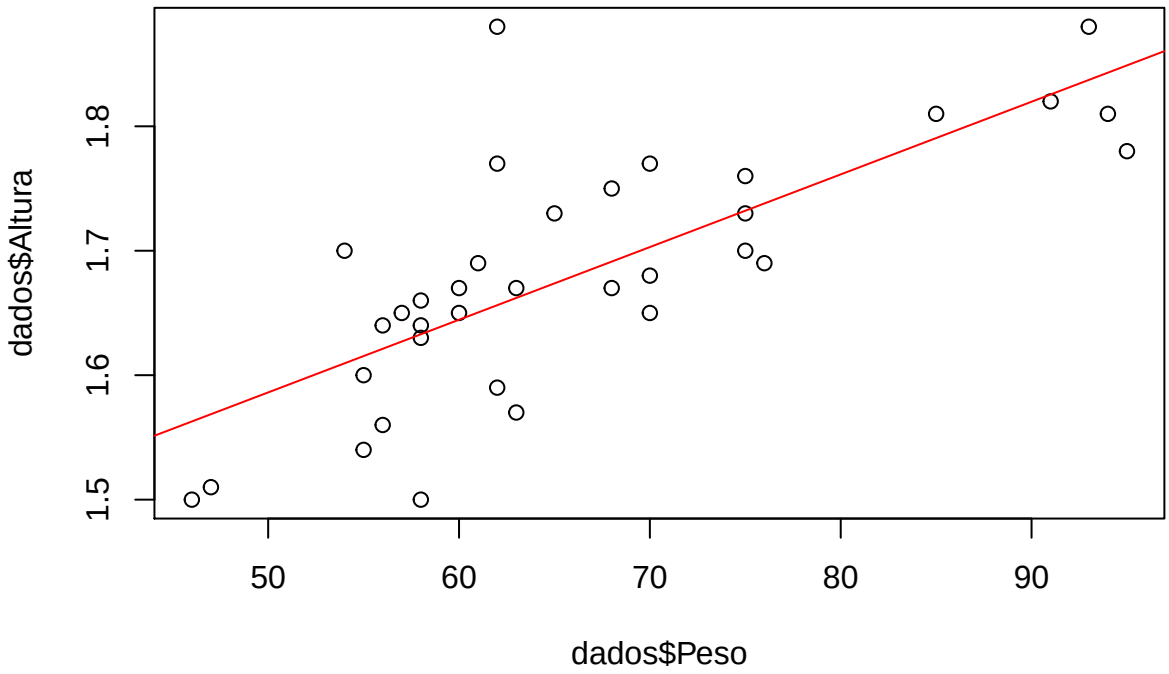
Se preferir, podemos usar um gráfico de densidades de Kernel como:

```r
d <- density(dados$Idade, na.rm = TRUE)

plot(d, col="blue", xlab="Idades",
     ylab="Densidade", main="Densidade das Idades")
```

Vamos explorar um pouco mais o gráfico de pontos. Sabe- se que existe uma correlação entre o peso e a altura das pessoas, e que, a distribuição dessas características configura uma bimodal se não considerarmos

os gênero dos indivíduos avaliados. Vamos avaliar a correlação entre essas caraterísticas com:

```
#Busque explorar mais parâmetros no gráfico a seguir:
##Dica: tente alterar o nome dos eixos
plot(x = dados$Peso, y= dados$Altura)

#Linha de ajuste
abline(lm(Altura ~ Peso, data = dados), col="red")
```




```
#Verificar a correlação:
cor(dados$Altura, dados$Peso)
```

```
## [1] 0.7480565
```

Alguns argumentos são específicos para cada função, busque mais informações sobre cada uma delas para entender sobre seus argumentos. Outros parâmetros gráficos mais recorrentes são os relacionados com:

- O tamanho de texto e símbolos, chamados de `cex`;
- Os símbolos de plotagem (25 ao todo), através do parâmetro `pch`;
- Há os relacionados com a fonte (`font` e `family`);
- Cores (`col`, `bg`, `fg`);

```
#Tamanho, pontos e cores
plot(x = dados$Peso, y= dados$Altura,
     main = "Peso  x Altura", #Título principal
     xlab = "Peso", #Eixo x
     ylab = "Altura", #Eixo y
     cex=2,
     cex.axis=1.5)

#Agora, busque alterar o nome dos rótulos (lab) e do título (main)
#Utilize o parâmetro 'pch=' e coloque um número de 0 a 25
#De maneira similar ao tamanho, trabalhe com as cores (utilize o color() para ver todas as cores dispon

# Fontes e linhas #
```

```r
plot(x = dados$Peso, y= dados$Altura,
     main = "Peso  x Altura", #Título principal
     xlab = "Peso", #Eixo x
     ylab = "Altura", #Eixo y
     font=3,
     family="serif", #Fonte do seu computador (Windows: "TT Courier New")
     )
abline(lm(Altura ~ Peso, data = dados), col="red", lty=3)


#Incrementando:
plot(x = dados$Peso, y= dados$Altura,
     main = "Peso  x Altura", #Título principal
     xlab = "Peso", #Eixo x
     ylab = "Altura", #Eixo y
     col = as.factor(dados$Genero)
     )

#Nesse mesmo gráfico, procure mudar a forma dos pontos de acordo com a nota do R.

#### Legenda ####
plot(x = dados$Peso, y= dados$Altura,
     main = "Peso  x Altura", #Título principal
     xlab = "Peso", #Eixo x
     ylab = "Altura", #Eixo y
     col = as.factor(dados$Genero),
     pch=10)

legend("bottomright", legend = c("Feminino", "Masculino"), col=1:2, pch=10)
```

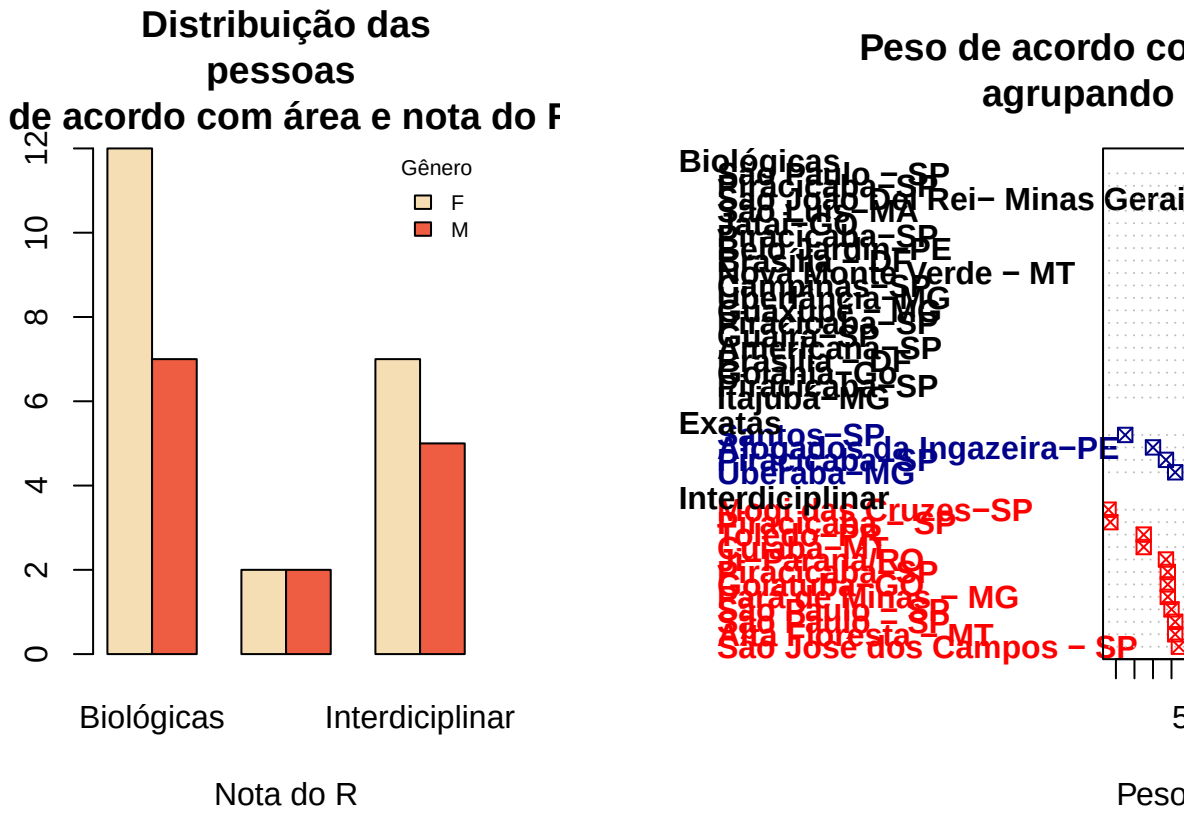
Obtidos todos esses gráficos, pode ser bom para visualização tê-los lado a lado.

```r
 #### Mais de um gráfico? Claro! ####
par(mfrow=c(1,2))
hist(dados$Altura)
hist(dados$Peso)

#Lembrar:
par(mfrow=c(1,1))
```

Desafio!!!

Busque resolver o desafio sem observar a resposta. Vocês conseguirão avançar muito, um ou outro detalhe pode acabar sendo mais difícil. Nesse ponto, vocês utilizam a resolução para entenderem melhor o como poderia ser feito. Veja que há muitos modos de resolver, apresentaremos apenas um deles.

- Dicas: 'beside', 'legend', 'bty = 'n'



Uma solução que utiliza tudo o que vocês aprenderam:

```
#1-) Fazer a divisão da tela:
par(mfrow=c(1,2))

#2-) Criar o barplot, da maneira que já aprendemos:
##Criar a tabela considerando as informacoes de Area e Genero
nota_gen <-table(dados$Area, dados$Genero)

##Note o parâmetro beside (que estava nas dicas)
barplot(t(nota_gen),beside = TRUE, col = c("wheat", "tomato2"), main = "Distribuição das
pessoas \n de acordo com área e nota do R", xlab = "Nota do R")
##Colocar a legenda no gráfico
legend("topright", title = "Gênero", legend=c("F","M"), fill = c("wheat", "tomato2"), bty = "n", cex = (

#3-) Criar o dotchart #Criar a coluna atribuindo uma cor para cada área
dados$color[dados$Area=="Exatas"] <- "darkblue"
dados$color[dados$Area=="Biologicas"] <- "darkgreen"
dados$color[dados$Area=="Humanas"] <- "orange"
dados$color[dados$Area=="Interdiciplinar"] <- "red"
dados$color[dados$Area=="Agrarias"] <- "black"

##Colocar os pesos em ordem:
dados <- dados[order(dados$Peso),]

##Colocar o título, note que o '\n' está sendo usado para fazer a quebra de linha:
dotchart(dados$Peso, labels = dados$Cidade, groups = as.factor(dados$Area),
         color = dados$color, font=2, main = "Peso de acordo com as cidades, \n agrupando por área",
```

```
         xlab = "Peso", pch = 7)

#4-) Manter novamente uma figura por gráfico
par(mfrow=c(1,1))
```

Salvar gráficos

Os gráficos podem ser salvos através dos menus disponíveis no RStudio, ou através de funções que permitem salvar em formatos específicos. Algumas delas são: pdf(); png(); jpeg(); bitmap(). De maneira geral, o parâmetro primordial é fornecer o nome do arquivo que será gerado (contendo sua extensão). Após abrir a função gráfica, deve-se gerar o gráfico de interesse. Por fim, utiliza-se o comando dev.off() para que saída gráfica volte para o console.

```
png(filename = "grafico_pontos.png")
plot(x = dados$Peso, y= dados$Altura,
     main = "Peso  x Altura", #Título principal
     xlab = "Peso", #Eixo x
     ylab = "Altura", #Eixo y
     col = dados$Genero,
     pch=10)
dev.off()

png(filename = "grafico_pontos.png", width = 1500, height = 1500, res= 300)
plot(x = dados$Peso, y= dados$Altura,
     main = "Peso  x Altura", #Título principal
     xlab = "Peso", #Eixo x
     ylab = "Altura", #Eixo y
     col = dados$Genero,
     pch=10)
dev.off()
```

Agora, gere um gráfico e salve-o no formato de seu interesse. Em seguida, crie diversos gráficos dentro de uma mesma função gráfica e estude a saída.

Instalação e aplicações de pacotes

Existem diversos pacotes disponíveis para variadas aplicações. Utilizaremos o *ggplot2*, que está disponível no repositório oficial do R, o CRAN. Portanto para instalá-lo:

```
install.packages("ggplot2")
```

Depois disso é necessário recrutá-lo com:

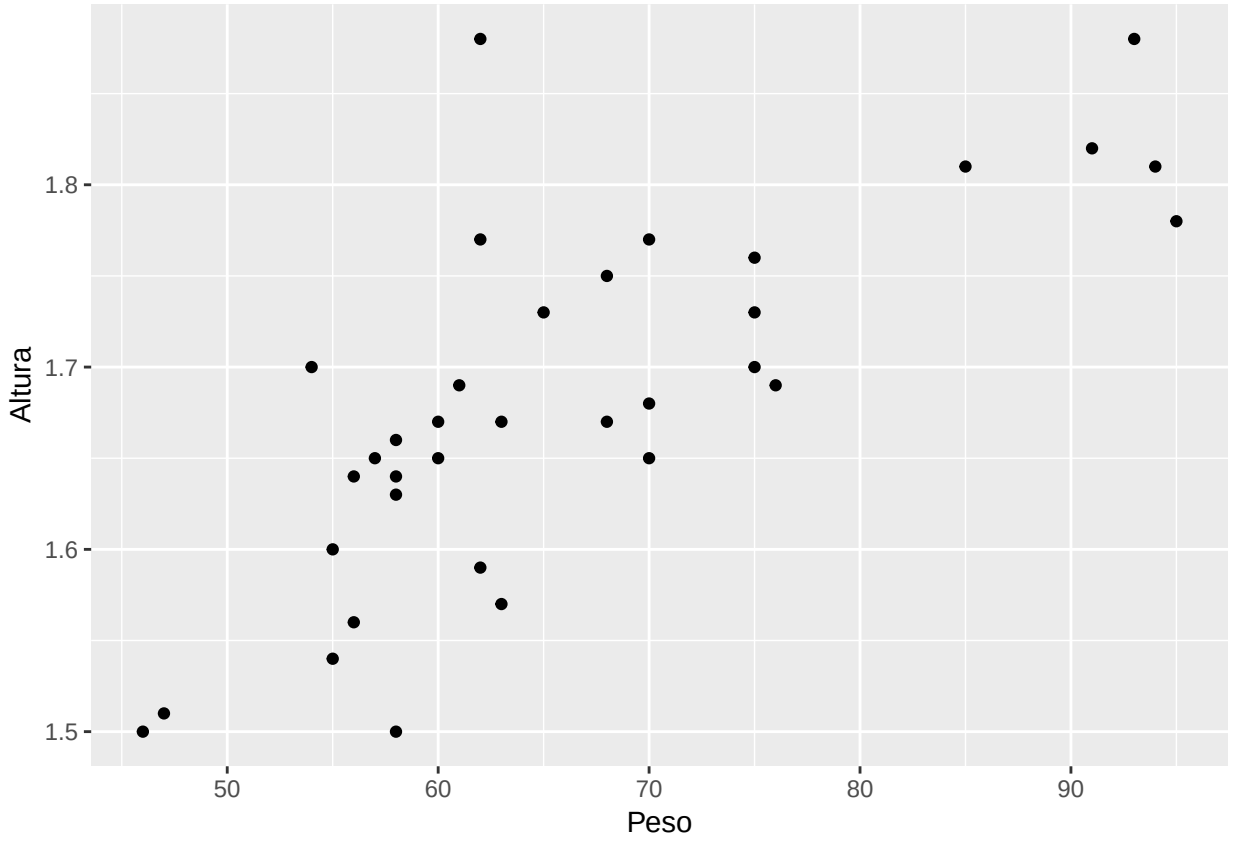
```
library("ggplot2")
```

```
## Registered S3 methods overwritten by 'ggplot2':
##   method         from
##   [.quosures     rlang
##   c.quosures     rlang
##   print.quosures rlang
```

O *ggplot2* é um pacote que permite a construção de gráficos estatísticos, suas funcionalidades vão muito além do que está disponível nos gráficos básicos do R. Vamos tentar fazer um exemplo?

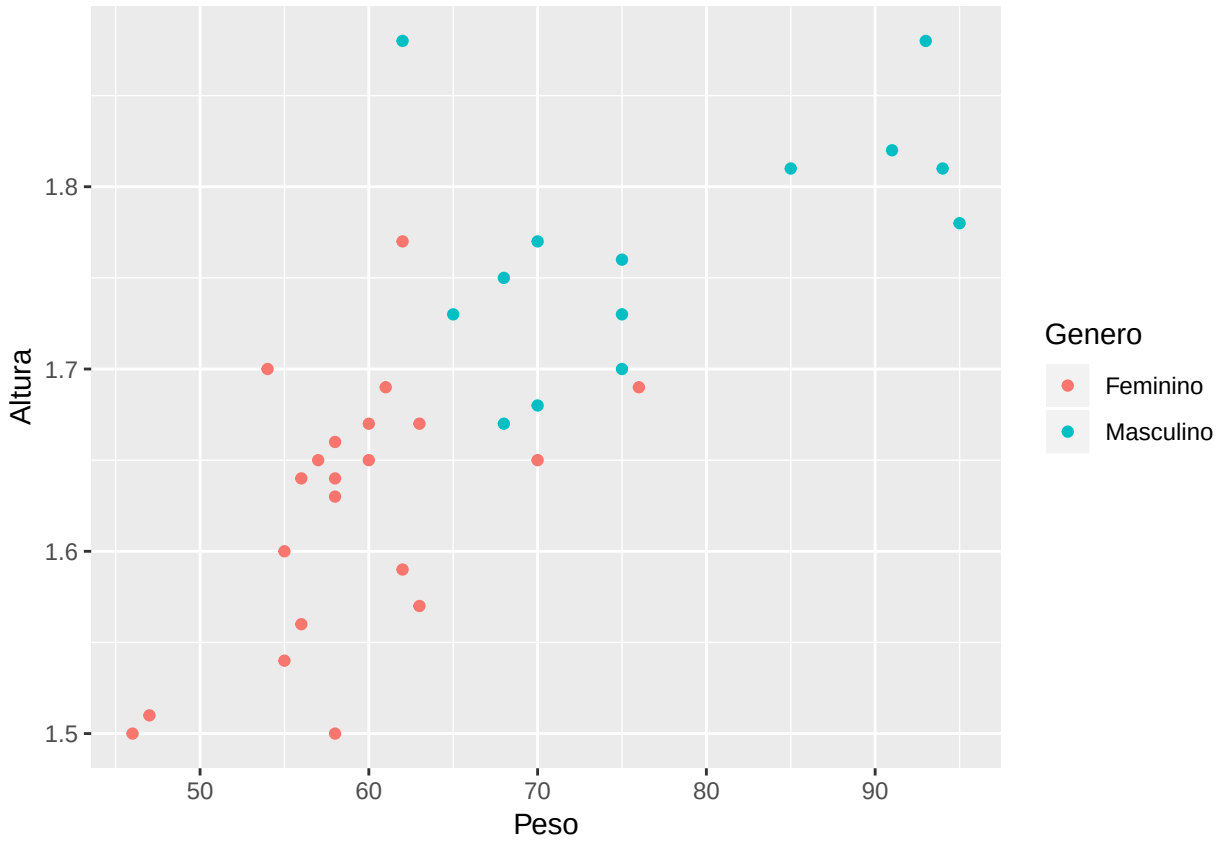
Primeiramente precisamos de um *data frame*, como o conjunto de dados que estamos usando as colunas para mapear as observações a serem dispostas no gráfico. Vamos criar um gráfico de dispersão com as variáveis *Peso* e *Altura*.

```
ggplot(dados) +
  geom_point(aes(x=Peso, y=Altura))
```



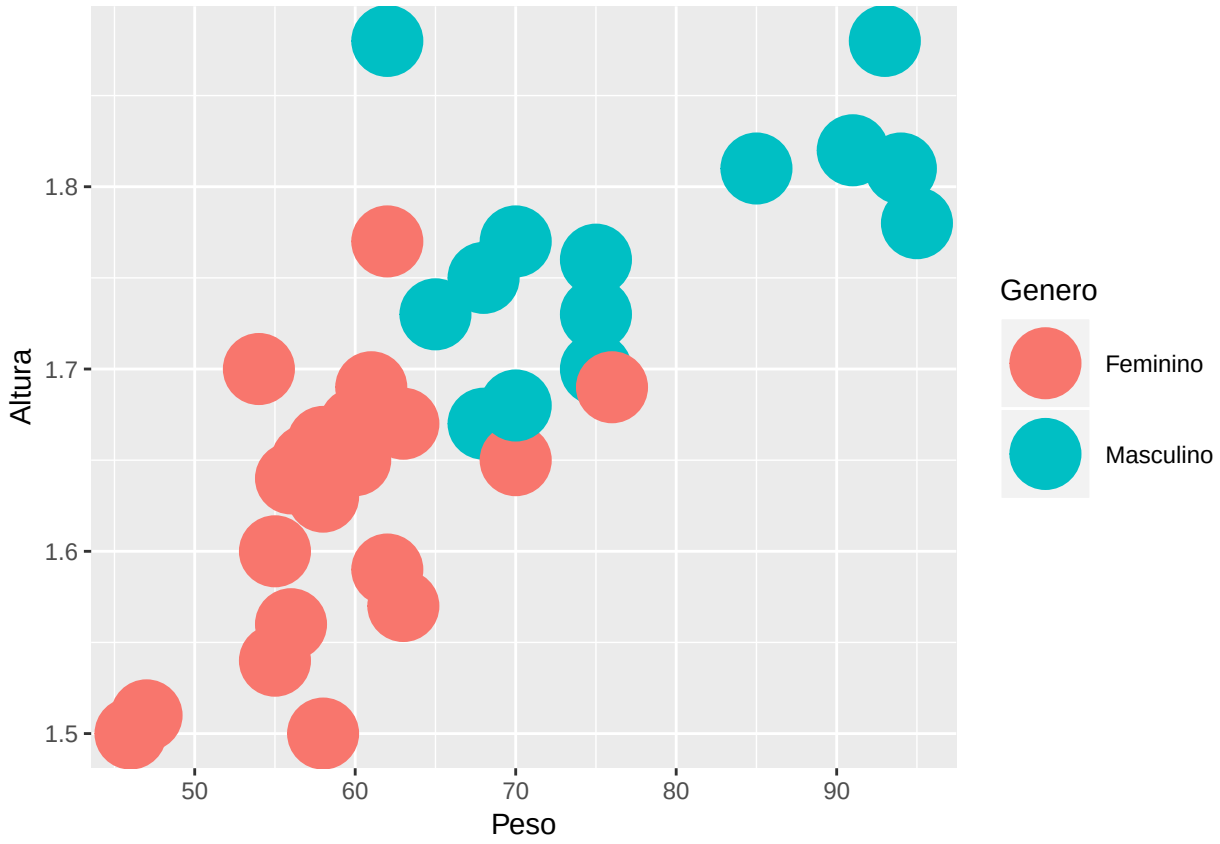
Veja que o sinal '+' indica que estamos adicionando uma nova camada. O legal do pacote é que a customização dos aspectos visuais é bem intuitiva. Por exemplo, vamos colorir os pontos de acordo com a variável *Genero*.

```
ggplot(dados) +
  geom_point(aes(x=Peso, y=Altura, colour=Genero))
```



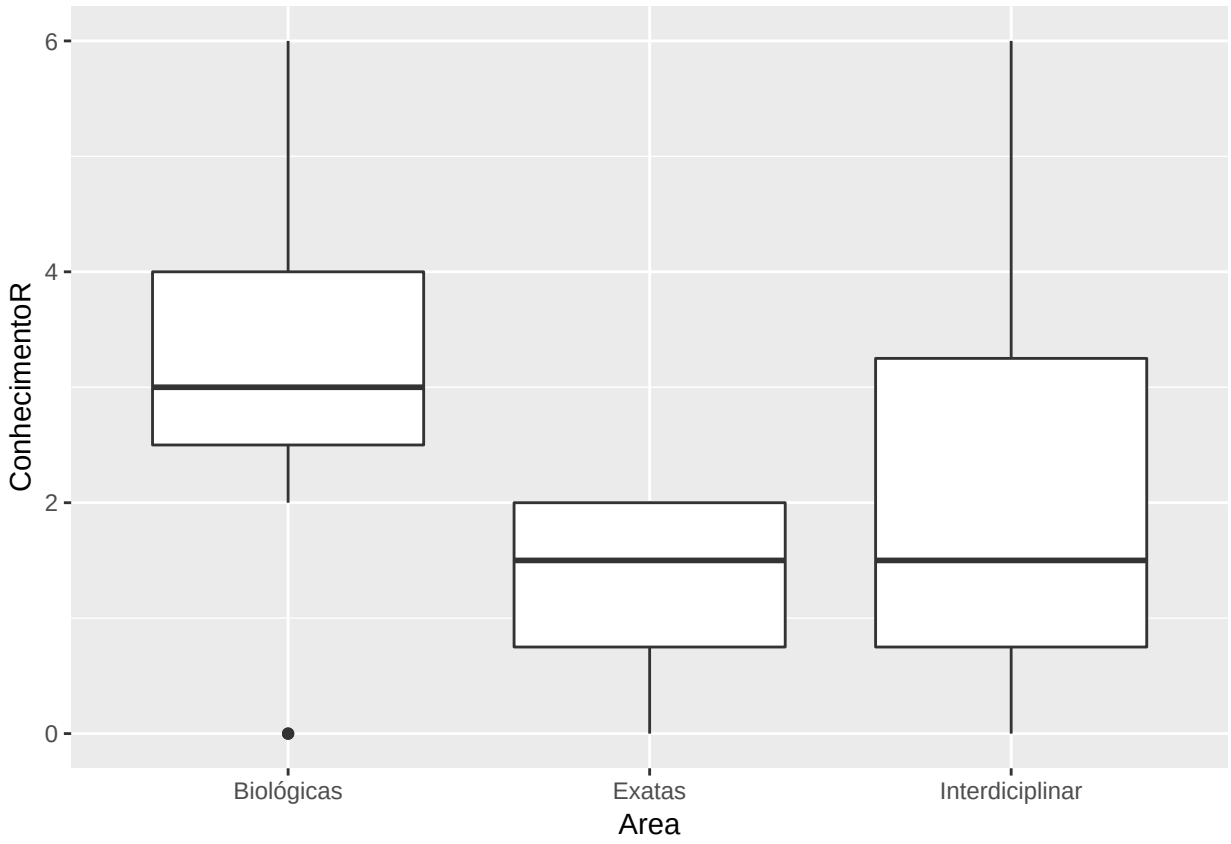
Também podemos alterar o tamanho dos pontos:

```
ggplot(dados) +
  geom_point(aes(x=Peso, y=Altura, color=Genero), size=12)
```

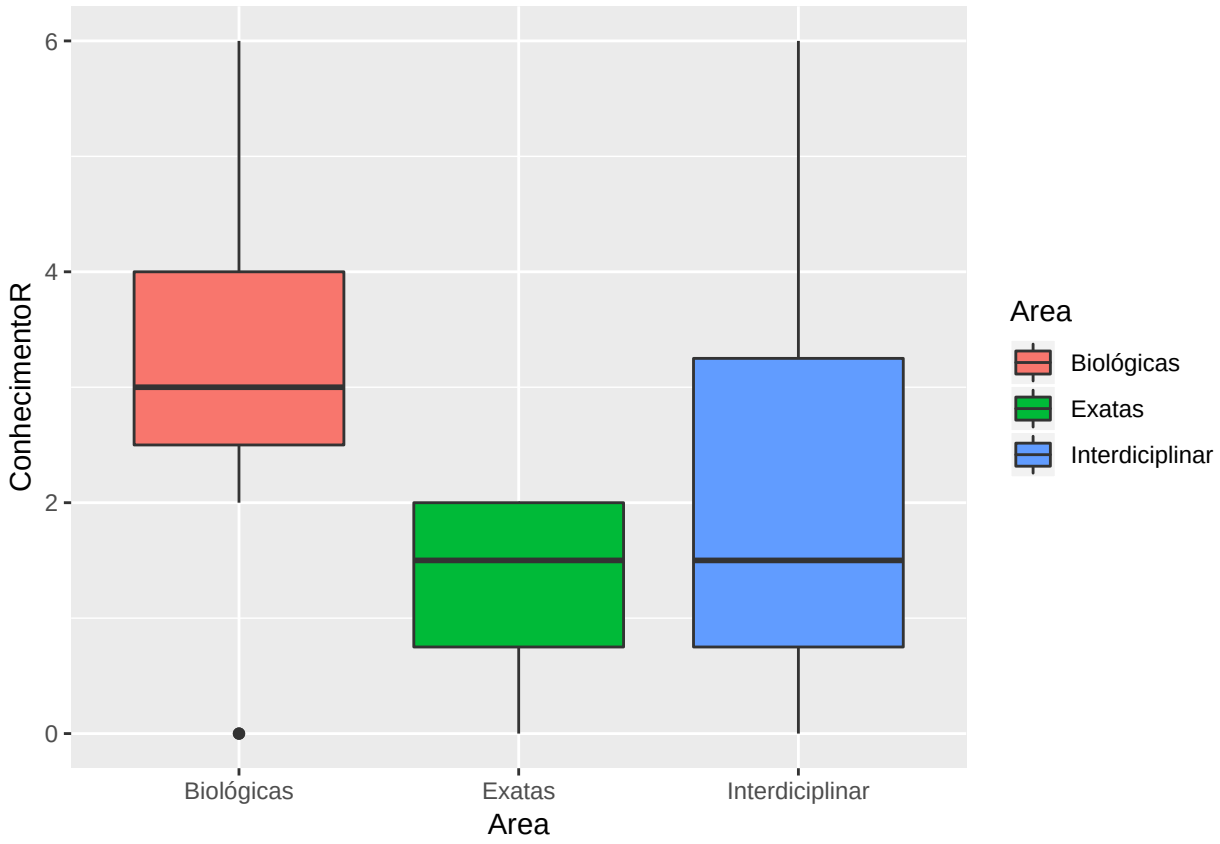


Veja que o parâmetro *size* permaneceu fora da função de mapeamento, *aes*. Lembre-se que o mapeamento precisa de variáveis com observações. Colocar o *size* apenas como parâmetro da *geom_point* é interessante porque vamos alterar o tamanho de todos os pontos do gráfico.

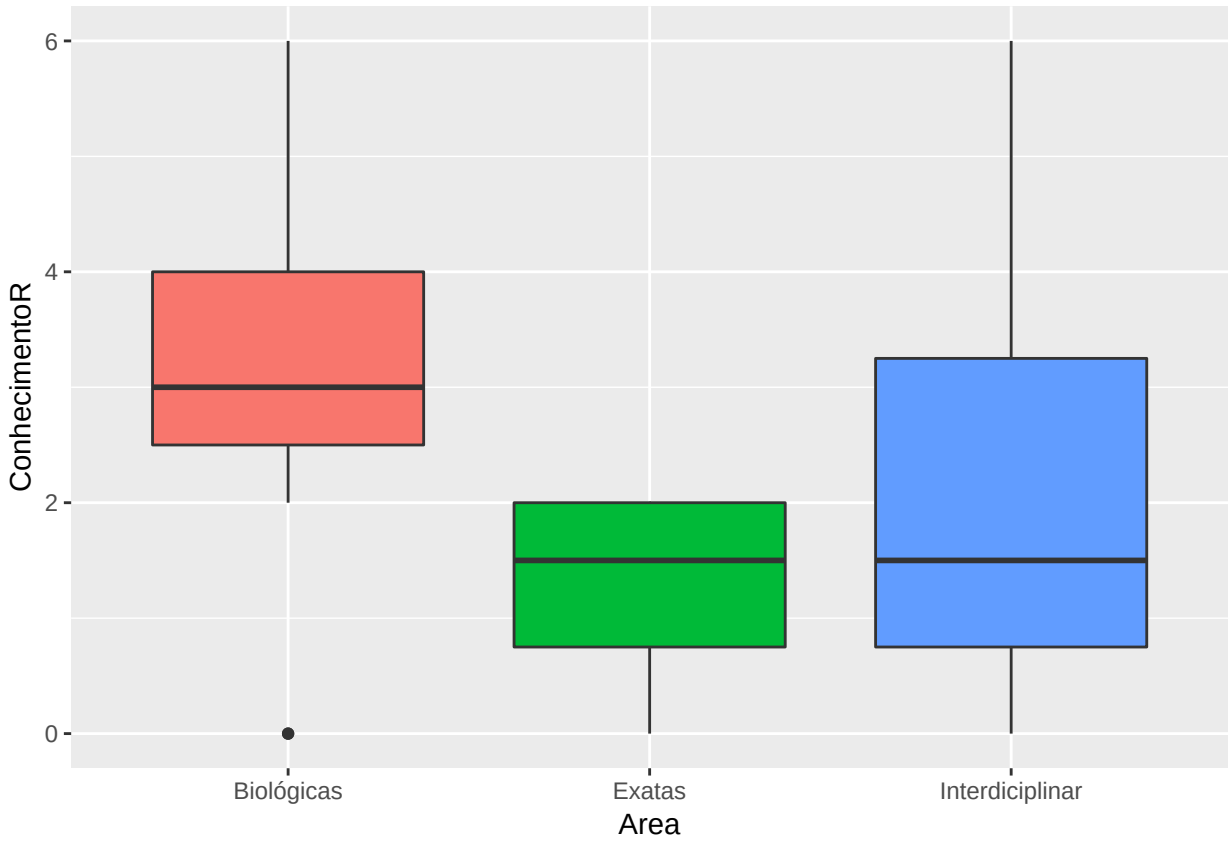
Que tal tentarmos fazer um boxplot utilizando como variável resposta o *ConhecimentoR* de acordo com a *Area*?



Quer botar uma corzinha? Tenta adicionar o *fill* de acordo com a *Area*.



Mas a legenda fica redundante com o eixo x, será que conseguimos suprimí-la? Tenta ver algo como *show.legend* na função *geom_boxplot*.



Paramos aqui na primeira parte da manhã de sábado (18.05) e seguimos para uma parte específica da sessão **Algumas Ferramentas de Análise de dados**, clique aqui para ir até lá.

E nossa motivação? Dá para fazer um gráfico?

Mas é claro que sim!!! Vai ser um pouquinho mais complexo, mas a gente chega lá.

Precisaremos instalar três pacotes: i) **tm**, ii) **SnowballC** e iii) **wordcloud**.

```
install.packages(c('tm', 'SnowballC', 'wordcloud'), dependencies = T)
```

```
library('tm')
library('SnowballC')
library('wordcloud')
```

Vamos ter que criar um Corpus e depois convertê-lo para um documento. Em seguida, removeremos a pontuação e as *stopwords*.

```
dataCorpus <- Corpus(VectorSource(dados$Motivacao))

dataCorpus <- tm_map(dataCorpus, content_transformer(tolower))
```

```
## Warning in tm_map.SimpleCorpus(dataCorpus, content_transformer(tolower)):
## transformation drops documents
```

```
dataCorpus <- tm_map(dataCorpus, removePunctuation)
```

```
## Warning in tm_map.SimpleCorpus(dataCorpus, removePunctuation):
## transformation drops documents
```

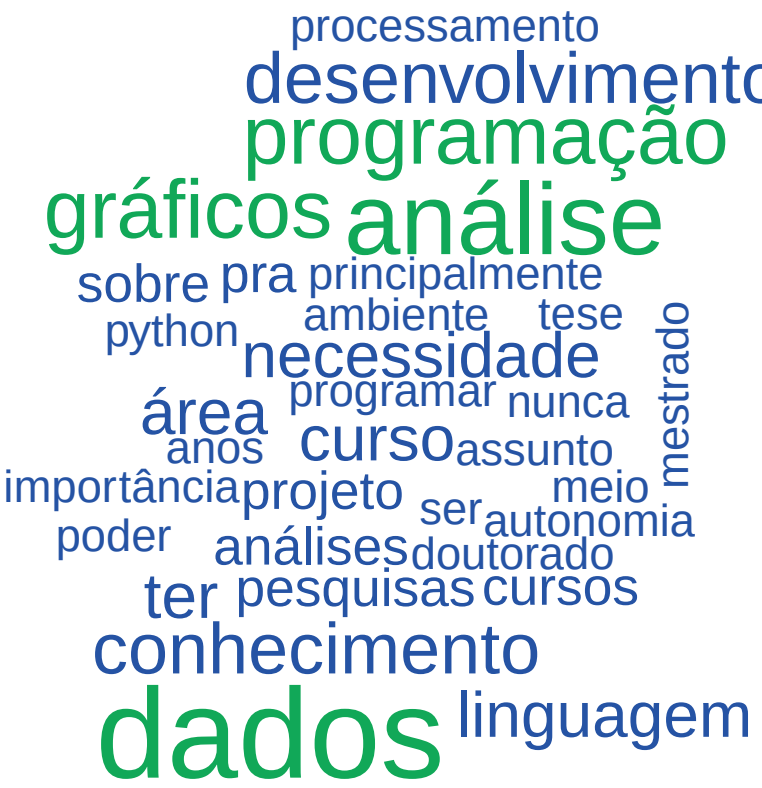
```
dataCorpus <- tm_map(dataCorpus, removeWords, stopwords('pt'))
```

```
## Warning in tm_map.SimpleCorpus(dataCorpus, removeWords, stopwords("pt")):
## transformation drops documents
```

Agora podemos construir nossa nuvem de palavras:

```
wordcloud(dataCorpus,max.words=100,colors=c("#2553A4","#11A858"))
```

```
## Warning in wordcloud(dataCorpus, max.words = 100, colors = c("#2553A4", :
## aprender could not be fit on page. It will not be plotted.
```

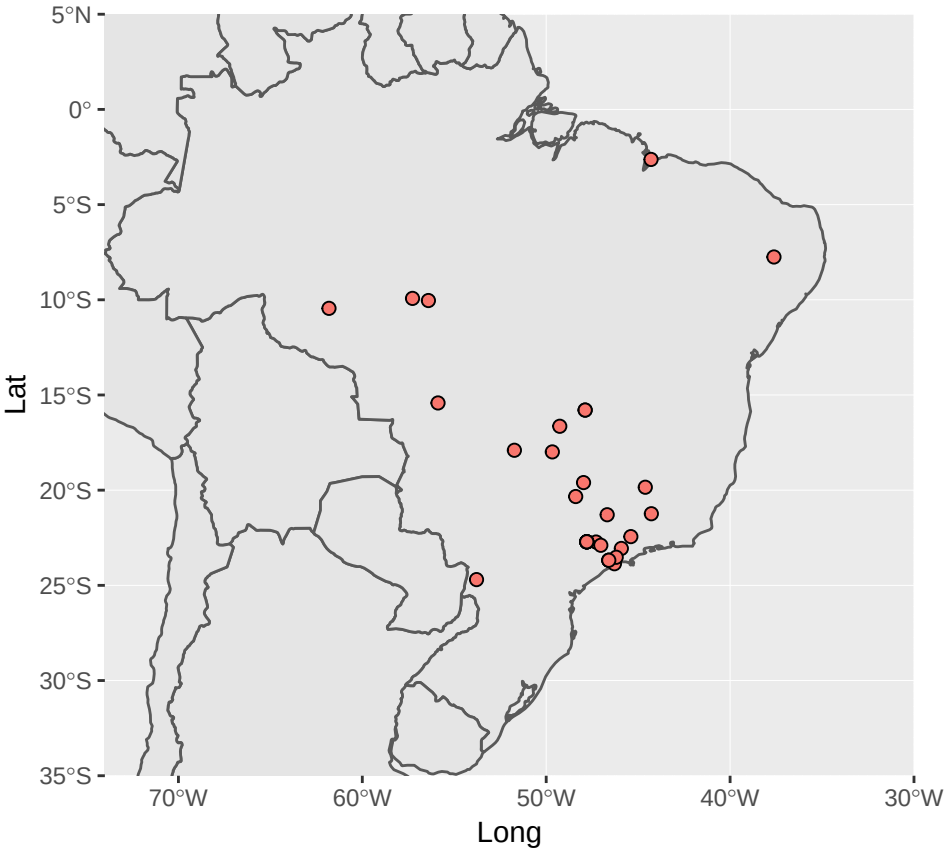


Mapa

Agora utilizaremos as informações de suas cidades para mostrá-las em um mapa. Vamos ter que instalar alguns pacotes, mas o resultado vale a pena:

```
# Carregar os pacotes
library("dplyr")
library("osmdata")
library("ggmap")
library("sf")
library("rnaturalearth")
library("rnaturalearthdata")
library("rgeos")

# Encontrar latitude e longitude das cidades
cidades <- data.frame(Long=NA, Lat=NA)
for(idx in 1:nrow(dados)){
  latLong <- getbb(paste0(dados$Cidade[idx]," ", "Brazil"))
```

```
  cidades[idx, ] <- rowMeans(latLong)

}

# Obter as fronteiras no mapa mundial
world <- ne_countries(scale = "medium", returnclass = "sf")

#Fazer o mapa
ggplot(data = world) +
    geom_sf() +
  geom_point(data = cidades,
             aes(x = Long, y = Lat, fill = "red"),
             size = 2, shape = 21, show.legend = FALSE) +
      coord_sf(xlim = c(-74, -30), ylim = c(-35, 5), expand = FALSE)
```



Família de funções `apply`

A família de funções `apply` também podem funcionar como um estrutura de repetição. Sua sintaxe é mais enxuta quando comparada com `for` ou `while` e pode facilitar a elaboração do código.

Aqui vamos exemplificar o uso de algumas dessas funções.

apply

A função `apply` é a base de todas as outras funções da família, portanto a compreensão do funcionamento desta é essencial para entender as demais. Se buscar no help da função, ele indicará que os argumentos da função consistem em: apply(X, MARGIN, FUN, ...). Sendo X o conjunto de dados em formato de array (incluindo matrix, que consiste num array de dimensão 2), MARGIN será 1 se a ação deverá ser aplicada à linhas, 2 se for aplicada a colunas e c(1,2) se for aplicada a ambas; FUN é a função que indica ação.

Num simples exemplo temos a matrix:

```
ex_mat <- matrix(seq(0,21,3), nrow = 2)
```

Se quisermos somar os elementos das colunas usamos:

```
apply(ex_mat, 2, sum)
```

```
## [1]  3 15 27 39
```

Se quisermos somar os elementos das linhas:

```
apply(ex_mat, 1, sum)
```

```
## [1] 36 48
```

Se fossemos utilizar o `for` para realizar essa tarefa:

```
# Soma das colunas
for(i in 1:dim(ex_mat)[2]){
  print(sum(ex_mat[,i]))
}
```

```
## [1] 3
## [1] 15
## [1] 27
## [1] 39
```

```
# Soma das linhas
for(i in 1:dim(ex_mat)[1]){
  print(sum(ex_mat[i,]))
}
```

```
## [1] 36
## [1] 48
```

lapply

Se diferencia do `apply` por poder receber outros tipos de objetos (mais utilizado com listas) e devolver o resultado em uma lista.

```
ex_list <- list(A=matrix(seq(0,21,3), nrow = 2),
                B=matrix(seq(0,14,2), nrow = 2),
                C= matrix(seq(0,39,5), nrow = 2))
str(ex_list)
```

```
## List of 3
##  $ A: num [1:2, 1:4] 0 3 6 9 12 15 18 21
##  $ B: num [1:2, 1:4] 0 2 4 6 8 10 12 14
##  $ C: num [1:2, 1:4] 0 5 10 15 20 25 30 35
```

Para selecionar a segunda coluna de todas as matrizes

```r
lapply(ex_list, "[", 2)
```

```
## $A
## [1] 3
##
## $B
## [1] 2
##
## $C
## [1] 5
```

sapply

A função `sapply` funciona como o `lapply` a diferença é que ele retorna apenas um valor por componente da lista e os deposita em um vetor de resposta. Como no exemplo:

```r
sapply(ex_list, "[",1,3)
```

```
##  A  B  C
## 12  8 20
```

tapply

Esta função é um pouco diferente das demais, ela exige que exista alguma variável categórica (fator) para aplicar ação separadamente conforme suas categorias (levels). Por isso, normalmente é aplicada a data.frames.

Vamos utilizar nosso conjunto de dados:

```r
str(dados)
```

```
## 'data.frame':    35 obs. of  13 variables:
##  $ Data_pesq        : chr  "5/17/2019 12:02:02" "5/17/2019 11:13:04" "5/17/2019 11:08:47" "5/17/2019
##  $ Idade            : int  29 26 25 23 35 26 34 35 27 24 ...
##  $ Niver            : chr  "19/06" "31/08" "25/09/1993" "02/09" ...
##  $ Genero           : chr  "Feminino" "Feminino" "Feminino" "Feminino" ...
##  $ Cidade           : chr  "Itajubá-MG" "Piracicaba-SP" "Goiania-Go" "Brasília - DF" ...
##  $ Altura           : num  1.5 1.51 1.7 1.54 1.6 1.56 1.64 1.65 1.5 1.64 ...
##  $ Peso             : int  46 47 54 55 55 56 56 57 58 58 ...
##  $ Area             : chr  "Biológicas" "Biológicas" "Biológicas" "Biológicas" ...
##  $ ConhecimentoR    : int  4 6 4 3 3 1 2 0 2 2 ...
##  $ Outras_linguagens: chr  "Não" "Linux" "NAO" "Não utilizo" ...
##  $ Utilizacao       : chr  "Análises e gráficos" "Análises dos resultados do meu projeto" "ESTATISTI
##  $ Motivacao        : chr  "Fazer gráficos de expressão relativa. E ter noções básicas do R" "Admiro
##  $ color            : chr  NA NA NA NA ...
```

```r
dados$Area <- as.factor(dados$Area)
tapply(dados$ConhecimentoR, dados$Area, mean)
```

```
##      Biológicas          Exatas Interdiciplinar
##        3.157895        1.250000        2.000000
```

Saiba mais sobre essa família de funções no link

Observe que nas funções `apply` podemos trocar as funções prontas do *r base* por funções personalizadas.

Faça o exercício **extra**

Algumas ferramentas básicas de análise de dados

Claramente a análise de dados é algo muito específico de cada conjunto de dados e interesses. Vamos aqui mostrar alguns recursos básicos como análise de variância, regressão e teste de médias.

Algumas avaliações descritivas podem ser feitas pelo uso do `tapply` e de gráficos. A função `summary` também da informações gerais do conjunto. É possível usá-la em conjunto com o `tapply`.

```
str(dados)
```

```
## 'data.frame':    35 obs. of  13 variables:
##  $ Data_pesq        : chr  "5/17/2019 12:02:02" "5/17/2019 11:13:04" "5/17/2019 11:08:47" "5/17/2019
##  $ Idade            : int  29 26 25 23 35 26 34 35 27 24 ...
##  $ Niver            : chr  "19/06" "31/08" "25/09/1993" "02/09" ...
##  $ Genero           : chr  "Feminino" "Feminino" "Feminino" "Feminino" ...
##  $ Cidade           : chr  "Itajubá-MG" "Piracicaba-SP" "Goiania-Go" "Brasília - DF" ...
##  $ Altura           : num  1.5 1.51 1.7 1.54 1.6 1.56 1.64 1.65 1.5 1.64 ...
##  $ Peso             : int  46 47 54 55 55 56 56 57 58 58 ...
##  $ Area             : Factor w/ 3 levels "Biológicas","Exatas",..: 1 1 1 1 1 3 1 1 1 3 ...
##  $ ConhecimentoR    : int  4 6 4 3 3 1 2 0 2 2 ...
##  $ Outras_linguagens: chr  "Não" "Linux" "NAO" "Não utilizo" ...
##  $ Utilizacao       : chr  "Análises e gráficos" "Análises dos resultados do meu projeto" "ESTATISTI
##  $ Motivacao        : chr  "Fazer gráficos de expressão relativa. E ter noções básicas do R" "Admiro
##  $ color            : chr  NA NA NA NA ...
```

```
# Certifique-se que esta lidando com variável categórica (fator)
dados$Genero <- as.factor(dados$Genero)

tapply(dados$Altura, dados$Genero, summary)
```

```
## $Feminino
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.500   1.570   1.640   1.623   1.670   1.770
##
## $Masculino
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.670   1.730   1.765   1.769   1.810   1.880
```

```
tapply(dados$Peso, dados$Genero, summary)
```

```
## $Feminino
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   46.00   56.00   58.00   58.81   62.00   76.00
##
## $Masculino
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   62.00   68.50   75.00   77.57   89.50   95.00
```

Podemos falar que os pesos são significativamente diferentes entre homens e mulheres?

```
mod1 <- lm(Peso ~ Genero, data = dados)
summary(mod1)
```

```
##
## Call:
## lm(formula = Peso ~ Genero, data = dados)
##
## Residuals:
```

```
##      Min       1Q   Median       3Q      Max
## -15.5714  -4.3095  -0.8095   3.6905  17.4286
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)       58.810      1.947  30.207  < 2e-16 ***
## GeneroMasculino   18.762      3.078   6.095  7.3e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8.922 on 33 degrees of freedom
## Multiple R-squared:  0.5296, Adjusted R-squared:  0.5153
## F-statistic: 37.15 on 1 and 33 DF,  p-value: 7.297e-07
```

```
# ou o equivalente

mod1 <- aov(Peso ~ Genero, data = dados)
summary(mod1)
```

```
##             Df Sum Sq Mean Sq F value  Pr(>F)
## Genero       1   2957  2956.9   37.15 7.3e-07 ***
## Residuals   33   2627    79.6
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

A diferença entre as duas funções é apenas na forma como é apresentada os resultados. O p-valor nos indica se podemos considerar diferenças do peso conforme o gênero.

Podemos obter mais informações sobre o modelo ajustado observando os gráficos:

```
plot(mod1)
```

Vamos utilizar outro conjunto de dados para realizarmos mais avaliações utilizando a função `lm`. Utilize:

```
load("clima_lond.RData")
```

Para obter os dados de precipitação da cidade de Londrina no primeiro semestre de 2017. Vamos utilizar as funções `tapply` e `lm` para avaliar os dados.

```
# Verificando se as variáveis categórias estão como fatores
str(clima_lond)
```

```
## 'data.frame':    181 obs. of  4 variables:
##  $ dia    : Factor w/ 31 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ Mes    : Factor w/ 6 levels "Abril","Fevereiro",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ prec.mm: Factor w/ 56 levels "0","0.2","0.3",..: 55 52 26 5 21 1 4 1 1 4 ...
##  $ Data   : Factor w/ 181 levels "2017-01-01","2017-01-02",..: 1 2 3 4 5 6 7 8 9 10 ...
```

```
clima_lond$dia <- as.factor(clima_lond$dia)

# A precipitação nesse caso é uma variável contínua, nao categórica, para transformá-la use:

clima_lond$prec.mm <- as.numeric(as.character(clima_lond$prec.mm))

# Já com o tapply podemos ver as diferenças

tapply(clima_lond$prec.mm, clima_lond$Mes, summary)
```

```
## $Abril
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.000   4.653   0.650  47.400
##
## $Fevereiro
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    0.00    0.00    3.65    2.85   34.80
##
## $Janeiro
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.600   8.532   6.600  59.600
##
## $Junho
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.000   3.573   0.500  30.600
##
## $Maio
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.000   7.406   7.000  48.200
##
## $Março
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.000   3.223   1.200  26.200
```

Repare que os levels aparecem em ordem alfabética e não conforme o tempo, alteramos isso com:

```
levels(clima_lond$Mes)
```

```
## [1] "Abril"     "Fevereiro" "Janeiro"   "Junho"     "Maio"      "Março"
```

```
pos <- match(c("Janeiro", "Fevereiro", "Marco", "Abril", "Maio", "Junho"), levels(clima_lond$Mes))
pos
```

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

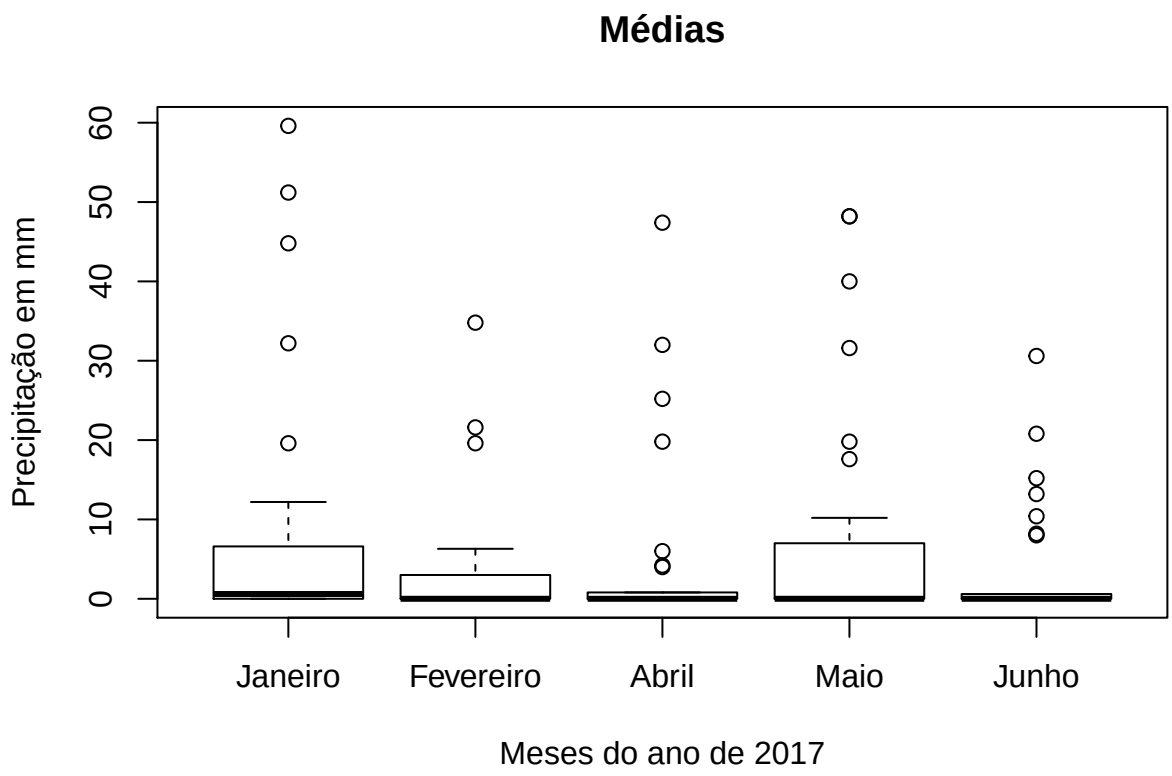
```
clima_lond$Mes = factor(clima_lond$Mes,
                        levels(clima_lond$Mes)[pos])

# Refazendo
tapply(clima_lond$prec.mm, clima_lond$Mes, summary)
```

```
## $Janeiro
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.600   8.532   6.600  59.600
##
## $Fevereiro
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    0.00    0.00    3.65    2.85   34.80
##
## $Abril
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.000   4.653   0.650  47.400
##
## $Maio
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.000   7.406   7.000  48.200
##
## $Junho
```

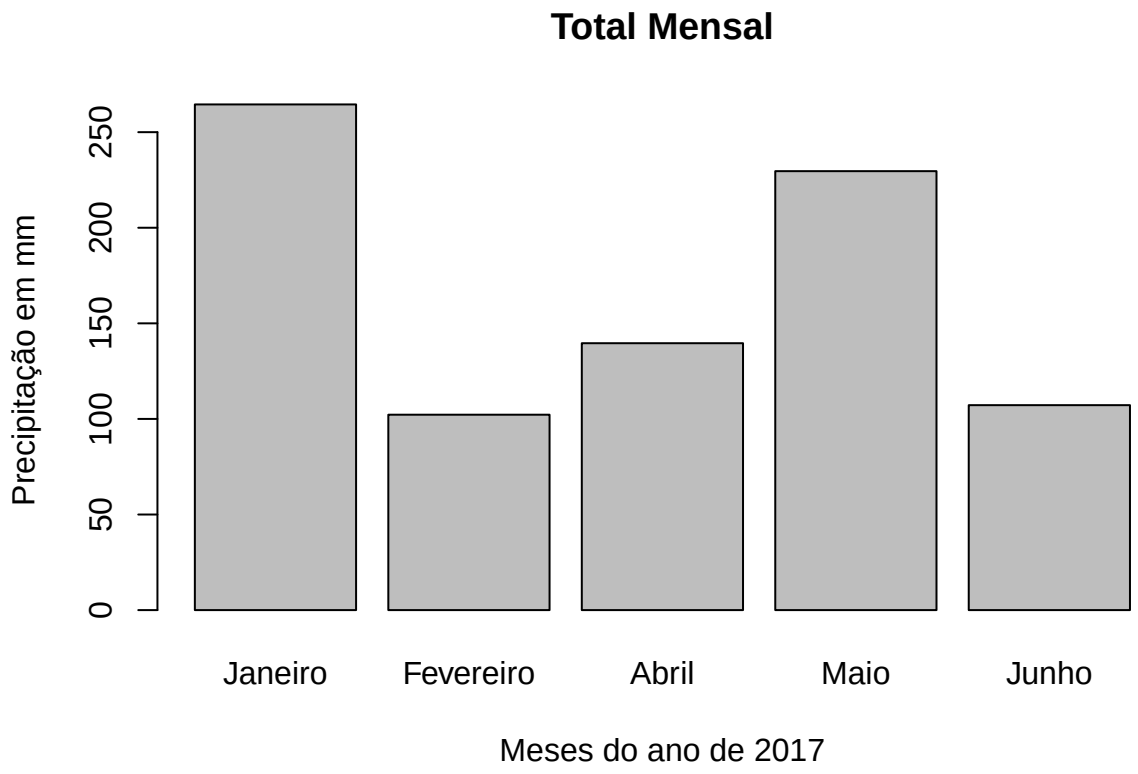
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.000   3.573   0.500  30.600
```

Podemos também avaliar fazendo alguns gráficos:

```
plot(prec.mm~Mes, data = clima_lond, main="Médias",
     xlab = "Meses do ano de 2017",
     ylab = "Precipitação em mm")
```




```
barplot(tapply(clima_lond$prec.mm, clima_lond$Mes, sum),
        main="Total Mensal",
        xlab = "Meses do ano de 2017",
        ylab = "Precipitação em mm")
```



Vamos então realizar um análise de variância para medir diferenças entre os meses.

```
mod <- lm(prec.mm ~ Mes, data = clima_lond)
summary(mod)
```

```
##
## Call:
## lm(formula = prec.mm ~ Mes, data = clima_lond)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -8.532 -7.406 -3.650 -1.125 51.068
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)     8.532      2.181   3.912  0.00014 ***
## MesFevereiro   -4.882      3.166  -1.542  0.12524
## MesAbril       -3.879      3.110  -1.247  0.21434
## MesMaio        -1.126      3.085  -0.365  0.71565
## MesJunho       -4.959      3.110  -1.594  0.11301
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12.14 on 145 degrees of freedom
##   (31 observations deleted due to missingness)
## Multiple R-squared:  0.02836,    Adjusted R-squared:  0.00156
## F-statistic: 1.058 on 4 and 145 DF,  p-value: 0.3795
```

Entre os dias:

```r
mod <- lm(prec.mm ~ dia, data = clima_lond)
summary(mod)
```

```
##
## Call:
## lm(formula = prec.mm ~ dia, data = clima_lond)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -16.317  -5.700  -1.117   0.000  46.067
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   10.633      4.680   2.272   0.0245 *
## dia2          -1.250      6.619  -0.189   0.8505
## dia3          -2.817      6.619  -0.426   0.6710
## dia4          -7.783      6.619  -1.176   0.2415
## dia5          -2.483      6.619  -0.375   0.7080
## dia6           5.683      6.619   0.859   0.3919
## dia7          -4.767      6.619  -0.720   0.4725
## dia8         -10.633      6.619  -1.607   0.1102
## dia9          -8.900      6.619  -1.345   0.1808
## dia10        -10.567      6.619  -1.597   0.1125
## dia11        -10.233      6.619  -1.546   0.1242
## dia12         -9.633      6.619  -1.455   0.1476
## dia13          1.050      6.619   0.159   0.8742
## dia14         -1.500      6.619  -0.227   0.8210
## dia15        -10.633      6.619  -1.607   0.1102
## dia16        -10.633      6.619  -1.607   0.1102
## dia17          2.900      6.619   0.438   0.6619
## dia18         -4.467      6.619  -0.675   0.5008
## dia19         -2.600      6.619  -0.393   0.6950
## dia20          2.033      6.619   0.307   0.7591
## dia21         -2.367      6.619  -0.358   0.7212
## dia22         -5.000      6.619  -0.755   0.4512
## dia23        -10.533      6.619  -1.591   0.1136
## dia24         -4.933      6.619  -0.745   0.4572
## dia25         -8.417      6.619  -1.272   0.2055
## dia26         -6.283      6.619  -0.949   0.3440
## dia27         -8.433      6.619  -1.274   0.2046
## dia28        -10.633      6.619  -1.607   0.1102
## dia29         -8.313      6.942  -1.198   0.2330
## dia30        -10.273      6.942  -1.480   0.1410
## dia31         -8.567      8.106  -1.057   0.2923
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11.46 on 150 degrees of freedom
## Multiple R-squared:  0.1666, Adjusted R-squared:  -7.178e-05
## F-statistic: 0.9996 on 30 and 150 DF,  p-value: 0.4755
```

Podemos também fazer um teste de médias para diferenciar a precipitação ao decorrer dos meses. Aqui utilizaremos o método de Tukey:

```r
modaov <- aov(prec.mm ~ Mes, data = clima_lond)
tukey.test <- TukeyHSD(x=modaov, 'Mes', conf.level=0.95)
tukey.test
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = prec.mm ~ Mes, data = clima_lond)
##
## $Mes
##                           diff        lwr       upr     p adj
## Fevereiro-Janeiro -4.88225806 -13.628251  3.863735 0.5369940
## Abril-Janeiro     -3.87892473 -12.470374  4.712524 0.7236982
## Maio-Janeiro      -1.12580645  -9.646543  7.394930 0.9961749
## Junho-Janeiro     -4.95892473 -13.550374  3.632524 0.5033691
## Abril-Fevereiro    1.00333333  -7.811566  9.818232 0.9978586
## Maio-Fevereiro     3.75645161  -4.989541 12.502445 0.7591298
## Junho-Fevereiro   -0.07666667  -8.891566  8.738232 0.9999999
## Maio-Abril         2.75311828  -5.838331 11.344567 0.9019673
## Junho-Abril       -1.08000000  -9.741585  7.581585 0.9969451
## Junho-Maio        -3.83311828 -12.424567  4.758331 0.7324452
```

Experimento de café

Agora, vamos trabalhar com outro conjunto de dados, contendo informações de um experimento de café. Acesse aqui:

- Arquivo cafe.txt

O experimento trata-se de dados em blocos casualizados de uma progênie de plantas de café. Nele contém uma coluna rep para especificar a qual repetição o dado se refere, outra para qual indivíduo da progênie (prog) e outra para indicar em qual colheita o dado foi retirado (colheita).

```r
data <- read.table("cafe.txt", h = TRUE, sep = "\t", dec = ",")
str(data)
```

```
## 'data.frame':    120 obs. of  4 variables:
##  $ rep     : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ prog    : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ colheita: int  1 1 1 1 1 1 1 1 1 1 ...
##  $ prod    : num  4.3 13.2 1.59 2.76 11.38 ...
```

Não esqueça que é necessário que o arquivo esteja no seu ambiente de trabalho ou que você especifique o caminho completo para que o R o encontre.

Para essa análise de dados, nossa variável resposta é a produção (prod), a repetição (rep), a progênie (prog) e a colheita serão fatores no nosso modelo, identificados por seus níveis.

```r
# Transformar em fator
data$rep <- as.factor(data$rep)
data$prog <- as.factor(data$prog)
data$colheita <- as.factor(data$colheita)
str(data)
```

```
## 'data.frame':    120 obs. of  4 variables:
##  $ rep     : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prog    : Factor w/ 10 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
```

```
##  $ colheita: Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prod    : num  4.3 13.2 1.59 2.76 11.38 ...
```

```
# Outra opção
data <- transform(data, rep = factor(rep), prog = factor(prog), colheita = factor(colheita))
str(data)
```

```
## 'data.frame':    120 obs. of  4 variables:
##  $ rep     : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prog    : Factor w/ 10 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ colheita: Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prod    : num  4.3 13.2 1.59 2.76 11.38 ...
```

Vamos primeiro análisar somente os dados referentes à primeira colheira. Podemos fazer um subset somente com esses dados.

```
# Indexar primeita colheita
Colheita_1 <- subset(data, colheita == 1)
str(Colheita_1)
```

```
## 'data.frame':    40 obs. of  4 variables:
##  $ rep     : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prog    : Factor w/ 10 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ colheita: Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prod    : num  4.3 13.2 1.59 2.76 11.38 ...
```

Repare que, ao fazer o subset, o conjunto de dados ainda mantém os três níveis do fator colheira, embora agora só tenhamos um. Isso pode ser um problema para a nossa análise, vamos remover os níveis excedentes com:

```
# Droplevels
Colheita_1 <- droplevels(subset(data, colheita == 1))
str(Colheita_1)
```

```
## 'data.frame':    40 obs. of  4 variables:
##  $ rep     : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prog    : Factor w/ 10 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ colheita: Factor w/ 1 level "1": 1 1 1 1 1 1 1 1 1 1 ...
##  $ prod    : num  4.3 13.2 1.59 2.76 11.38 ...
```

Agora podemos rodar nosso modelo de análise de variância.

```
# Modelo
Modelo1 <- aov(prod ~ rep + prog,
               contrasts = list(prog = "contr.sum"),
               data = Colheita_1)
anova(Modelo1)
```

```
## Analysis of Variance Table
##
## Response: prod
##           Df Sum Sq Mean Sq F value   Pr(>F)
## rep        3  58.90  19.633  2.1836 0.113071
## prog       9 410.32  45.591  5.0708 0.000475 ***
## Residuals 27 242.75   8.991
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Essa análise variância exige alguns pressupostos, podemos verificar eles nos nossos dados usando:

```
#####################################################
###verificar Pressupostos da análise de variância###
#####################################################
names(Modelo1)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "contrasts"     "xlevels"       "call"          "terms"
## [13] "model"
```

```
Modelo1_residuals <- Modelo1$residuals #armazenando os erros ou resíduos

# teste de Normalidade DOS ERROS##
#---------------------------------#
shapiro.test (Modelo1_residuals) # Hipótese de Nulidade
```

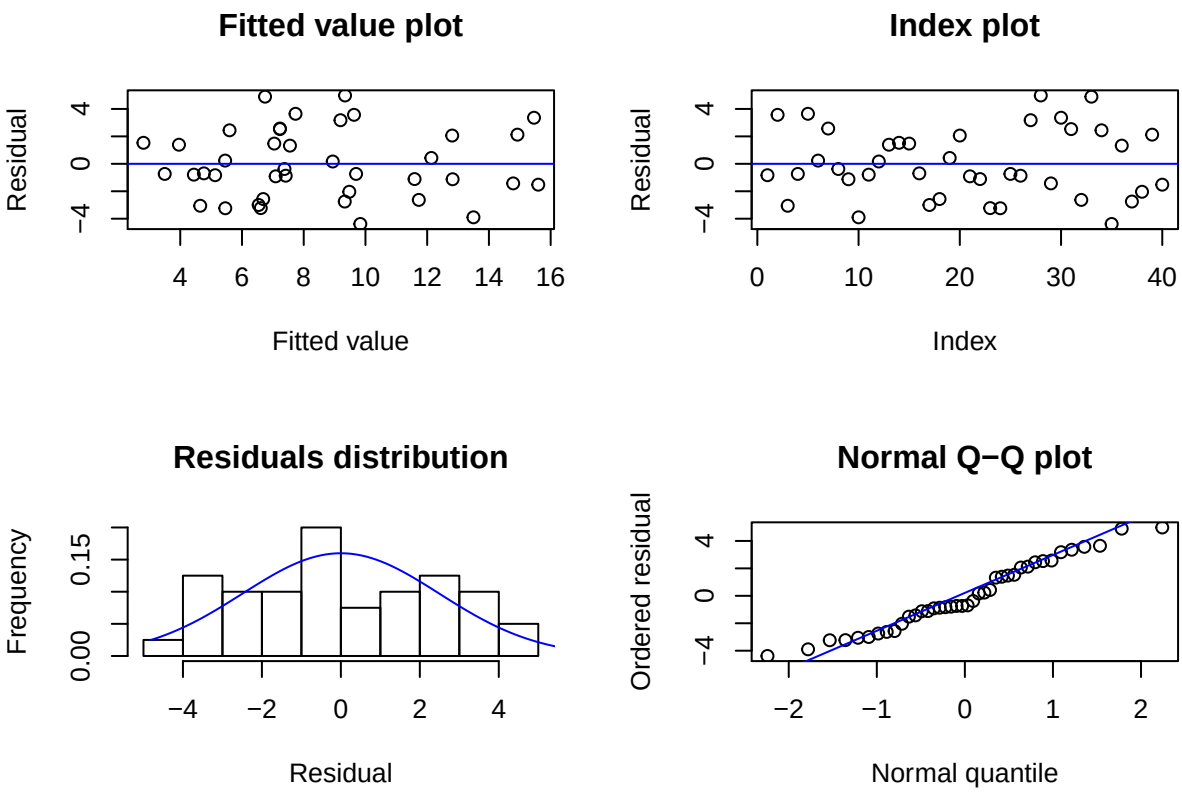
```
##
##  Shapiro-Wilk normality test
##
## data:  Modelo1_residuals
## W = 0.96494, p-value = 0.2461
```

```
# a hipótese de que os erros são normais, nesse caso, como o p-value = 0.24
# ou seja, >0.05 ou 0.01 ou qualquer alfa adotado, não se rejeita a hipotese de normalidade
```

Agora, vamos precisar de funções contidas nos scripts `DIAGNOSTICS.R` e `"outliers.R"`. Acesse eles em:

- DIAGNOSTICS.R
- outliers.R

```
# Funções
source("DIAGNOSTICS.R")
source("outliers.R")


#Verificar pressupostos de análises
diagnostics(Modelo1$residuals, Modelo1$fitted.values)
```



As vezes outliers podem ser difíceis de identificarmos, aqui esta uma função que pode fazer isso:

```
# Verificar outlier
outlier(Modelo1$residuals, alpha=0.05)
```

```
## No outlier detected
```

```
## numeric(0)
```

Guardando o valor do quadrado médio:

```
QME <- anova(Modelo1)["Residuals", "Mean Sq"]
QME
```

```
## [1] 8.990907
```

E a média total da produçao:

```
med <- mean(data$prod, na.rm = TRUE)
med
```

```
## [1] 8.12025
```

Com eles podemos calcular o coeficiente de variação (CV):

```
CVe <- (sqrt(QME)/med)*100
CVe
```

```
## [1] 36.92601
```

Calcule o CVe e QME para a colheita 2

Crie uma função calcular o CVe

Possibilidade de respostas:

```r
CV_E <- function(anova, med){
  QME <- anova(anova)["Residuals", "Mean Sq"]
  CVe <- (sqrt(QME)/med)*100

  return(CVe)
}

##
CV_E(anova = Modelo1, med = med)
```

```
## [1] 36.92601
```

Podemos também calcular a herdabilidade da característica produção:

```r
n_rep <- nlevels(Colheita_1$rep)
VG <- (anova(Modelo1)["prog", "Mean Sq"]- QME)/n_rep
VE <- QME
H_2 <- VG/ (VG + VE)
H_2
```

```
## [1] 0.5043871
```

Crie uma função para estimar a herdabilidade para o DBC

Agora faremos um teste de comparações múltiplas utilizando a análise de variância feita no pacote `ExpDes`.

```r
#install.packages("ExpDes")
library(ExpDes)
rbd(Colheita_1$prog, Colheita_1$rep, Colheita_1$prod, mcomp = "tukey", sigT = 0.05, sigF = 0.05)
```

```
## ------------------------------------------------------------------------
## Analysis of Variance Table
## ------------------------------------------------------------------------
##            DF     SS     MS     Fc    Pr>Fc
## Treatament  9 410.32 45.591 5.0708 0.000475
## Block       3  58.90 19.633 2.1836 0.113071
## Residuals  27 242.75  8.991
## Total      39 711.97
## ------------------------------------------------------------------------
## CV = 35.09 %
##
## ------------------------------------------------------------------------
## Shapiro-Wilk normality test
## p-value:  0.2460709
## According to Shapiro-Wilk normality test at 5% of significance, residuals can be considered normal.
## ------------------------------------------------------------------------
##
## ------------------------------------------------------------------------
## Homogeneity of variances test
## p-value:  0.2852398
## According to the test of oneillmathews at 5% of significance, the variances can be considered homoce
## ------------------------------------------------------------------------
##
## Tukey's test
## ------------------------------------------------------------------------
## Groups Treatments Means
## a      10       14.3425
```

```
## a     9   13.6675
## ab    2   10.4725
## ab    5   8.5775
## ab    8   8.22
## ab    7   8.0725
##  b    6   6.2975
##  b    1   5.9725
##  b    3   5.49
##  b    4   4.34
## ------------------------------------------------------------------------
```

```
# Teste Sk
rbd(Colheita_1$prog, Colheita_1$rep, Colheita_1$prod, mcomp = "sk", sigT = 0.05, sigF = 0.05)
```

```
## ------------------------------------------------------------------------
## Analysis of Variance Table
## ------------------------------------------------------------------------
##            DF     SS     MS     Fc    Pr>Fc
## Treatament  9 410.32 45.591 5.0708 0.000475
## Block       3  58.90 19.633 2.1836 0.113071
## Residuals  27 242.75  8.991
## Total      39 711.97
## ------------------------------------------------------------------------
## CV = 35.09 %
##
## ------------------------------------------------------------------------
## Shapiro-Wilk normality test
## p-value:  0.2460709
## According to Shapiro-Wilk normality test at 5% of significance, residuals can be considered normal.
## ------------------------------------------------------------------------
##
## ------------------------------------------------------------------------
## Homogeneity of variances test
## p-value:  0.2852398
## According to the test of oneillmathews at 5% of significance, the variances can be considered homoce
## ------------------------------------------------------------------------
##
## Scott-Knott test
## ------------------------------------------------------------------------
##    Groups Treatments   Means
## 1       a         10 14.3425
## 2       a          9 13.6675
## 3       a          2 10.4725
## 4       b          5  8.5775
## 5       b          8  8.2200
## 6       b          7  8.0725
## 7       b          6  6.2975
## 8       b          1  5.9725
## 9       b          3  5.4900
## 10      b          4  4.3400
## ------------------------------------------------------------------------
```

```
# função by é similar ao subset
by(data, data$colheita, function (x) anova(aov(prod ~ prog + rep, data= x)))
```

```
## data$colheita: 1
```

```
## Analysis of Variance Table
##
## Response: prod
##           Df Sum Sq Mean Sq F value   Pr(>F)
## prog       9 410.32  45.591  5.0708 0.000475 ***
## rep        3  58.90  19.633  2.1836 0.113071
## Residuals 27 242.75   8.991
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## ------------------------------------------------------------
## data$colheita: 2
## Analysis of Variance Table
##
## Response: prod
##           Df  Sum Sq Mean Sq F value  Pr(>F)
## prog       9 143.327 15.9253  3.0702 0.01147 *
## rep        3  12.757  4.2524  0.8198 0.49430
## Residuals 27 140.049  5.1870
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## ------------------------------------------------------------
## data$colheita: 3
## Analysis of Variance Table
##
## Response: prod
##           Df  Sum Sq Mean Sq F value  Pr(>F)
## prog       9 307.340  34.149  2.9481 0.01421 *
## rep        3   9.104   3.035  0.2620 0.85213
## Residuals 27 312.752  11.583
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
#verificar os graus de liberdade, cuidado para que sejam utilizados de fato os dados de apenas 1 colhei


#outra opção selecionando apenas a parte da dataframe onde colheita igual a "2"
anova(aov(data$prod ~ data$prog + data$rep, data = data[data$colheita==2,]))
```

```
## Analysis of Variance Table
##
## Response: data$prod
##            Df  Sum Sq Mean Sq F value  Pr(>F)
## data$prog   9  293.00  32.556  2.5748 0.01007 *
## data$rep    3   44.42  14.808  1.1711 0.32428
## Residuals 107 1352.91  12.644
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
# mais uma opção
indiv <- lapply(split(data, data$colheita), aov, formula = prod ~ rep + prog)
lapply(indiv, summary)
```

```
## $`1`
##             Df Sum Sq Mean Sq F value   Pr(>F)
## rep          3   58.9   19.63   2.184 0.113071
## prog         9  410.3   45.59   5.071 0.000475 ***
```

```
## Residuals   27  242.8    8.99
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## $`2`
##             Df Sum Sq Mean Sq F value Pr(>F)
## rep          3  12.76   4.252    0.82 0.4943
## prog         9 143.33  15.925    3.07 0.0115 *
## Residuals   27 140.05   5.187
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## $`3`
##             Df Sum Sq Mean Sq F value Pr(>F)
## rep          3    9.1    3.03   0.262 0.8521
## prog         9  307.3   34.15   2.948 0.0142 *
## Residuals   27  312.8   11.58
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Agora, com o pacote **agricolae** elaboraremos o sorteio de um experimento também de blocos casualizados.

```
############################
#SORTEIO DE EXPERIMENTOS####
############################

#install.packages("agricolae")
library(agricolae)
trt <- c("0","1","2","5","10","20","50","100","Dina")
rcbd <- design.rcbd(trt, 6, serie = 1, seed = 1, "default") # seed = 1
rcbd # Planilha de campo
```

```
## $parameters
## $parameters$design
## [1] "rcbd"
##
## $parameters$trt
## [1] "0"    "1"    "2"    "5"    "10"   "20"   "50"   "100"  "Dina"
##
## $parameters$r
## [1] 6
##
## $parameters$serie
## [1] 1
##
## $parameters$seed
## [1] 1
##
## $parameters$kinds
## [1] "default"
##
## $parameters[[7]]
## [1] TRUE
##
##
```

```
## $sketch
##      [,1]   [,2] [,3]   [,4] [,5]   [,6]  [,7]   [,8]   [,9]
## [1,] "Dina" "5"  "50"   "0"  "1"    "20"  "2"    "100"  "10"
## [2,] "1"    "2"  "100"  "0"  "10"   "20"  "Dina" "50"   "5"
## [3,] "50"   "0"  "Dina" "10" "20"   "100" "5"    "1"    "2"
## [4,] "10"   "1"  "100"  "20" "0"    "5"   "2"    "Dina" "50"
## [5,] "2"    "20" "1"    "50" "5"    "10"  "100"  "Dina" "0"
## [6,] "50"   "20" "0"    "5"  "Dina" "2"   "1"    "100"  "10"
##
## $book
##    plots block  trt
## 1     11     1 Dina
## 2     12     1    5
## 3     13     1   50
## 4     14     1    0
## 5     15     1    1
## 6     16     1   20
## 7     17     1    2
## 8     18     1  100
## 9     19     1   10
## 10    21     2    1
## 11    22     2    2
## 12    23     2  100
## 13    24     2    0
## 14    25     2   10
## 15    26     2   20
## 16    27     2 Dina
## 17    28     2   50
## 18    29     2    5
## 19    31     3   50
## 20    32     3    0
## 21    33     3 Dina
## 22    34     3   10
## 23    35     3   20
## 24    36     3  100
## 25    37     3    5
## 26    38     3    1
## 27    39     3    2
## 28    41     4   10
## 29    42     4    1
## 30    43     4  100
## 31    44     4   20
## 32    45     4    0
## 33    46     4    5
## 34    47     4    2
## 35    48     4 Dina
## 36    49     4   50
## 37    51     5    2
## 38    52     5   20
## 39    53     5    1
## 40    54     5   50
## 41    55     5    5
## 42    56     5   10
## 43    57     5  100
```

```
## 44    58     5 Dina
## 45    59     5    0
## 46    61     6   50
## 47    62     6   20
## 48    63     6    0
## 49    64     6    5
## 50    65     6 Dina
## 51    66     6    2
## 52    67     6    1
## 53    68     6  100
## 54    69     6   10
```

Podemos exportar e salvar nosso sorteio com:

```
write.table(rcbd,"SORTEIO.txt", row.names=FALSE, sep="\t")
file.show("SORTEIO.txt")
write.csv(rcbd,"SORTEIO.csv",row.names=F)
```

Paramos aqui no sábado (18.05),terceiro e último dia do treinamento.

Pratique gerando relatórios no RStudio

Utilize o R no seu dia-a-dia para ir praticando a linguagem. Além das recomendações contidas na primeira apresentação, recomendamos também dar uma olhada em como gerar documentos em pdf e html usando a Markdown. Utilizamos essa metodologia para gerar este tutorial e outras apresentações do curso. Pode ser muito prático no dia-a-dia!

Para utilizar, será necessário a instalação de outros pacotes. Um deles é o próprio `rmarkdown`:

```
install.packages("rmarkdown")
```

```
library(rmarkdown)
```

Agora crie um arquivo .Rmd utilizando as facilidades do RStudio, clique no ícone com símbolo `+` no canto superior esquerdo. Escolha o opção `R Markdown`. Dê um título ao seu arquivo e escolha a opção `html`. Ao fazer isso, o RStudio já coloca um template inicial, ja com um cabeçalho:

```
---
title: "Teste"
author: "Eu"
date: "June 5, 2018"
output: html_document
---
```

Este é o mais simples possível, você pode otimizá-lo de diversas maneiras. Saiba mais aqui.

O template inicial também traz alguns exemplos de sintaxe do markdown. Observe que utilizando `#` para títulos de sessões, `##` para um nível inferior (subtitulos) e assim por diante. Palavras em negrito são escritas em meia a dois `*` e existem diversas outras especificações para essa sintaxe. Veja mais sobre ela aqui.

Para compilar o código, basta clicar em `Knit`. Ele irá pedir para que o arquivo .Rmd seja salvo com algum nome em algum lugar.

O markdown também é capaz de entender diretamente a linguagem html, também a css e latex. Para essa última, o latex precisa estar instalado e todas suas dependências.

Existem alguns pacotes que fornecem templates mais robustos para produção de htmls. Para esse tutorial utilizando o pacote `rmdformats` e personalizamos suas cores. Experimente:

```r
install.packages("rmdformats")
```

Agora faça o mesmo procedimento, clique no `+`, escolha `R Markdown` e, antes de escolher um título, mude para `From Template`, escolha o `HTML readthedown template`. Copie e cole o seguinte texto e aperte `Knit`.

```
# Teste1

Isso aqui é um teste só para dar uma olhada no template

## Testinho

Subsessão

* Item

**negrito**

*itálico*

fiz um [link](https://GENt-esalq.github.io/)!
```

Saiba mais no tutorial sobre isso no R-bloggers, que acreditamos ser um bom começo! Acesse aqui.

Caso tenha sugestões para aprimoramento desse material, enviar e-mail para `gent.esalq@gmail.com`.

Acesse também outros materiais em português produzidos por Cristiane Taniguti, Fernando Correr e Rodrigo Amadeu aqui.

Este material foi produzido por alunos do programa de pós-graduação em Genética e Melhoramento de Plantas. Cristiane Taniguti, Fernando Correr e Kaio Olimpio ministraram o Treinamento. Também contamos com os monitores: Letícia de Castro Lara, Getúlio Caixeta, Gabriel Gesteira, Rafael Yassue, Fernando Espolador, Júlia Morosini, Guilherme Hokasa, Vitor Mello, Ana Letycia Basso, Jessica Ferrarezi e Jéssica Nogueira de Souza.

Também recomendamos materiais em inglês aqui.