Programação no R & Python

Gustavo Eduardo Marcatti
Departamento de Engenharia Florestal
Universidade Federal de São João del-Rei

Abril 2021

Contents

PREFÁCIO

Falar sobre o livro aqui.

Falar sobre os tópicos a seguir... introduzir sobre os tópicos...

Capítulo 1

POR QUE PROGRAMAR?

Falar sobre o livro aqui.

Falar sobre os tópicos a seguir... introduzir sobre os tópicos...

Automatizar tarefas: Executar sequência, muitas vezes longa e complexa, de instruções via comando único. Fator velocidade: Computador é capaz de executar cálculos com elevada precisão e rapidez, superando, e muito, a capacidade do ser humano. Atualmente, o fator velocidade é essencial, pois a cada dia são coletados volumes de dados cada vez maiores, o que torna praticamente impossível a derivação de informações sem o auxílio do computador, além disso, as informações devem ser geradas e disseminadas em um intervalo de tempo cada vez menor. Sem a comunicação em tempo hábil, tal informação pode perder o sentido e ser pouco útil para os processos de tomada de decisão.

Fator erro: Minimiza a quantidade de trabalhos repetitivos e monótonos, diminuindo as chances de ocorrer erros humanos devido ao cansaço e redução gradativa da concentração.

Pensamento criativo vs rotineiro: A programação possibilita ao profissional dedicar mais tempo para o pensamento criativo, que demanda capacidade de imaginação e percepção. Já o pensamento rotineiro requer pouco talento, a não ser aquele de seguir instruções corretamente. Independência de softwares já prontos: Todos os softwares são concebidos para resolver um conjunto de problemas que surgem com maior frequência no dia-a-dia dos usuários, pelo menos de acordo com as observações, impressões e pesquisa do desenvolvedor frente ao seu nicho de mercado. É ilusório acreditar que existe softwares desenvolvidos para resolver todos os seus problemas. É muito comum se deparar com problemas sem soluções implementadas em softwares, em alguns casos o trabalho é interrompido ou então a estratégia de resolução deve ser alterada. Ou altera-se o algoritmo ou então utiliza-se procedimentos manuais.

Documentar metodologias: A sequência de instruções é armazenada em um documento de forma lógica e intuitiva, podendo ser consultada e facilmente entendida, mesmo após anos do desenvolvimento. Especialmente se junto com os códigos alguns comentários, para descrever as etapas mais críticas, forem adicionados.

Não se sentir um peixe fora d'água: Dentro do seu ambiente de trabalho, ser capaz de identificar problemas que são passíveis de serem resolvidos via programação e resolver por conta própria ou então buscar equipe (internamente ou externamente) capacitada para resolver tal problema.

Potencializa a capacidade de resolver problemas: Essa é uma opinião compartilhada por muitos programadores. A prática de programação desenvolve a capacidade de pensar de forma sis-

temática e objetiva, o que facilita o processo de resolução de problemas variados, inclusive da vida pessoal. Aprender a programar permite organizar melhor as ideias, focar no que é mais importante e crítico. De maneira geral o conhecimento de programação permite ver o mundo de outra forma. Em muitos países, a prática de programação é incentivada, e faz parte do plano pedagógico de muitas escolas. O incentivo começa com as crianças, mesmo em idades iniciais. A programação favorece a aprendizagem, sobretudo em disciplinas relacionados às ciências exatas. A seguir é apresentada uma frase dita pelo Steve Jobs: "Todas as pessoas deveriam aprender a programar um computador, pois isso ensina você a pensar".

Segurança para resolver problemas: Alguns dos problemas que enfrentamos no dia a dia não possuem solução implementada disponível, diante desse fato, o caminho a ser seguido seria alterar a metodologia de resolução, o que nem sempre é possível ou então buscar profissional qualificado no mercado, o que pode ser difícil ou então muito oneroso. Ainda existe o cenário mais pessimista, que é simplesmente a constatação de que o problema não pode ser resolvido com a atual infraestrutura em mãos. O fato principal aqui é que com um certo conhecimento de programação dificilmente será intimidado pelos problemas apresentados.

Menos estresse: Com softwares de aponte-e-clique momentos de estresse podem ser mais frequente, sobretudo porque esse tipo de software é muito mais suscetível a bugs do que uma linguagem de programação .Isso porque com esses softwares, além de se preocupar com a implementação do algoritmo que efetivamente resolve o problema é necessário desenvolver uma interface gráfica para o usuário, algo muito complexo de ser feito. Bugs são coisas

realmente estressantes, às vezes a ferramenta não entrega o que promete ou então causa a interrupção do funcionamento do software. Aqui cabe um parêntese, de acordo com minha experiência, na maioria das vezes que esses problemas acontecem é culpa do usuário. Isso porque muitas vezes o usuário não sabe "pedir", isto é, não executa a ferramenta correta ou então não executa a correta como se deveria (como instruído no manual). Ok, mas o desenvolvedor deveria prever isso e simplesmente emitir uma mensagem de erro, com uma instrução para executar a função corretamente. Geralmente os desenvolvedores fazem isso, e softwares já maduros fazem isso com maestria, inclusive com planejamento de interfaces que minimizam as chances de o usuário fazer algo errado. Mas imaginar que um software irá controlar (cercar) todas as chances de nós cometermos erros é completamente ilusório. Além disso, geralmente quanto maior é o controle mais engessado é o software.

Maior satisfação pessoal: Nada melhor para autoestima do que resolver um problema programação computacional. Você se sente parte da solução, com um sentimento que sua participação foi vital para resolver o problema. Possibilidade de estar na fronteira do conhecimento: É muito comum que uma pesquisa inovadora necessite de determinado tipo específico análise, mas pode ser que simplesmente não existe metodologia publicada sobre a análise e muito menos implementações disponíveis, nesses casos a programação pode ser útil, pois o atraso na publicação pode significar a perda do caráter inovador e até mesmo de uma eventual patente.

Confere capacidade de identificar e utilizar códigos prontos disponíveis: Existe quantidade muito grande de códigos

disponíveis para resolver uma infinidade de problemas. Porções de códigos são muito mais simples de serem desenvolvidos do que softwares com interface gráfica com o usuário. Seja um reciclador de códigos, evite reinventar a roda!

Aprender sobre temas complexos: Alguns temas, especificamente relacionados à matemática, são indigestos para boa parte dos estudantes. Essa dificuldade está associada a diversos fatores, mas a ausência de aplicações práticas com explicação do procedimento de forma detalhada, pode ser apontando como um dos principais. Com a programação é possível realizar experimentos práticos de forma simples e rápida, além disso, com a programação é possível explorar os conceitos mais básicos da matemática, que apesar de básicos os estudantes têm dificuldades de entender com aulas puramente teóricas, isso devido à baixa capacidade de abstração.

Aprender a utilizar a matemática de forma "correta": A resolução de problema matemático requer 4 etapas básicas: (1) Identificar as questões corretas / definir o problema: requer proatividade e conhecimento técnico sobre o tema de estudo; (2) Formular o problema: converter um problema do mundo real em uma formulação matemática, e se possível já na forma de código para ser resolvido por um computador. É aqui que o conhecimento de programação é importante e útil; (3) Encontrar a solução: tarefa executada por um computador, seria basicamente fazer contas; (4) Avaliação da solução: converter um problema matemático em uma solução passível de ser executada no mundo real. Também é a etapa em que a decisão é tomada.

Se preparar para o futuro: Na verdade se preparar para o agora! A demanda por programadores já é elevada e a tendên-

cia é aumentar cada vez mais, atualmente muitas empresas demandam profissional com o domínio de programação além da formação técnica convencional, tal como agronomia, engenharia civil, engenharia florestal e biologia. Há previsões mais extremistas de que a programação será tão importante e necessária quanto disciplinas básicas, tais como biologia e física, e até mesmo uma atividade tão básica quanto dirigir um carro. Apesar dessas previsões serem um tanto exageradas o que não resta dúvida é que a programação está se tornando cada vez mais relevante e seu domínio pode se tornar a alavanca que você precisa para alcançar colocações melhores no mercado.

Capítulo 2

POR QUE O R?

O R é um software ou linguagem de programação? R é um ambiente completo de desenvolvimento: é um ambiente integrado de funções para manipulação de dados, cálculos e gráficos; além de um conjunto completo de estruturas de controle (condicional e repetição).

É Gratuito: o R pode ser copiado e distribuído entre os usuários, bem como pode ser instalado em diversos computadores livremente, promovendo uma economia para empresas e pessoas físicas, devido ao não pagamento de taxas de licenças que são cobradas por outros softwares pagos, que além de serem altas são bem restritivas.

Facilidade de uso: Apesar do R ser executado a partir de comandos, não é necessário ser um programador para aproveitar dos benefícios oferecidos, pois uma grande quantidade de rotinas já estão implementadas, se o usuário não encontrar determinada função que execute a análise requerida, esta pode ser criada com

certa facilidade, pelo menos comparativamente com outras linguagens de programação.

Facilidade de criar novos procedimentos: o R possui uma linguagem de programação bem desenvolvida, simples e efetiva, que inclui condicionais, estruturas de repetição, funções recursivas definidas pelo usuário, e facilidades para entrada e saída de dados. O R ainda suporta a vetorização, o que permite executar procedimentos repetitivos (loops) sem a necessidade de definição explícita.

Compartilhamento: Facilidade e rapidez de troca de informações e conhecimentos, pois análises complexas podem ser realizadas com poucas linhas de comando, que na verdade são essencialmente blocos de textos. Esses textos poderão ser enviados ou recebidos através de um simples e-mail ou mensagem de WhatsApp, ou então acessadas em salas virtuais de grupos de ajuda por pesquisa em sites de busca, como o Google. Em contrapartida, para compartilhar análise análoga em um software com interface de aponte-e-clique o gasto de energia será muito maior, exigindo uma série de capturas de tela (print screen) e setinhas indicando o significado de cada elemento da janela do software.

Executar análises complexas: Possibilidade de executar análises complexas de forma simples nas mais variadas áreas do conhecimento. Abaixo é apresentado o ajuste de modelos utilizando duas estratégias completamente distintas, com elevada variação em termos de complexidade, porém a forma de declarar esses modelos no R são bem semelhantes. Primeiro o ajuste de uma regressão linear múltipla, utilizando a função lm (Fitting Linear Models) da biblioteca stats; em seguida o ajuste via redes neurais artificiais, utilizando a função nnet (Fit Neural

Networks) da biblioteca de mesmo nome.

```r
# Dados: y em função de x1 e x2
y <- c(0.21, 0.25, 0.1, 0.79, 0.55, 0.39, 0.71)
x1 <- c(0.51, 0.66, 0.9, 0.05, 0.42, 0.7, 0.33)
x2 <- c(0.1, 0.23, 0.15, 0.9, 0.65, 0.44, 0.81)

# Regressão Linear Múltipla
linear_multipla <- lm(y ~ x1 + x2)

# Rede Neural Artificial
library(nnet) # Carregar biblioteca nnet
# 2 neurônios na camada oculta
rede_neural <- nnet(y ~ x1 + x2, size = 2)
```

Código fonte aberto: Permite o acesso à rotina utilizada em determinada análise, possibilitando a alteração do código de acordo com necessidades específicas do usuário e possibilita aprender o princípio de funcionamento de determinado procedimento via exame do código. Além disso, as falhas podem ser detectadas com maior facilidade, e as correções e atualizações poderão ser disponibilizadas em questões de dias pelo grupo que gerencia o R (Core Development Team).

Quantidade de extensões: O R pode ser estendido via funções, scripts e principalmente via criação de novas bibliotecas (ou pacotes). O R possui uma infinidade de bibliotecas para as mais variadas áreas do conhecimento (ver CRAN Task Views).

Capacidade gráfica: É possível construir gráficos variados, robustos e com elevada qualidade tipográfica de forma simples e

rápida. O R é recomendado para confecção final de figuras para livros e materiais didáticos devido à sua qualidade tipográfica.

Multiplataforma: R é disponível para muitas plataformas, incluindo Unix, Linux, Windows, Macintosh.

Disponibilidade de materiais de apoio: existência de inúmeros manuais, tutoriais, cadernos didáticos, apostilas e livros destinados a ensinar o uso do R.

O RStudio: plataforma de desenvolvimento madura, amplamente utilizada pela comunidade e com um excelente editor de texto, que conta com uma série de funcionalidades, tais como identificação de erros de sintaxe; complemento de funções e objetos; coloração diferenciada de objetos e estruturas de controle; atalhos de teclado úteis, como o de executar códigos (ctrl + Enter) e os de alterar do editor para o console (ctrl + 2) e do console para o editor (ctrl + 1); comando para endentação automática; além de outras funcionalidades, como janelas específicas para plotar figuras, acessar arquivos e bases de dados, consultar os documentos de ajuda das funções.

Muitas possibilidades de fazer a mesma coisa: positivo, mas pode ser negativo, sobretudo para iniciantes; dica: identificar os pacotes/autores de confiança, evitar usar um novo pacote para executar determinados procedimentos que podem ser executados com a combinação de poucas funções de um pacote básico. **Curva de aprendizagem íngreme (negativo) vs flexibilidade e capacidade de resolver problemas:** mais flexível do que ambientes aponte-e-clique. Mesmo assim, aprender R é muito mais fácil do que uma série de outras linguagens de programação. **Ausência de assistência técnica formal (negativo):**

o grupo que gerencia o R não se responsabiliza pelos resultados retornados pela execução das rotinas disponibilizadas, além de não ofertar suporte técnico formalmente vs comunidade ativa de usuários e suporte técnico via contratação de terceiros.

Capítulo 3

INICIANDO NO R

3.1 Comandos e ambiente de trabalho

Comandos no R são expressões inseridas no prompt `>` e finalizadas com a tecla Enter, é executado um comando. O prompt já apresentando automaticamente e indica que o R está pronto para receber um comando[1], se o prompt for digitado junto com o comando, uma mensagem de erro será emitida. Cada linha representa um comando, alternativamente pode-se inserir vários comandos em uma mesma linha, porém cada comando deve estar separado dos demais por ponto e vírgula `;`.

Para executar o primeiro comando é necessário fazer o download e instalação do software R no Desktop. O download pode ser feito no site https://www.r-project.org/ e o usuário deve se atentar para escolher a alternativa de acordo com seu sistema op-

[1]Neste momento, uma descrição sucinta de vetor será feita. No capítulo sobre estrutura de dados, uma descrição mais completa será apresentada, também serão apresentadas outras estruturas.

eracional, uma vez que o R é multiplataforma, versões para Windows, Mac e diversas distribuições Linux são disponibilizadas.

Após a instalação, o R pode ser inicializado nas versões 32 bits (R i386) ou 64 bits (R x64). Atualmente a maioria dos processadores e sistemas operacionais são de 64 bits, então preferencialmente opte pela versão de 64 bits, até mesmo porque na versão de 32 bits existe um limite teórico de endereçamento de $232 = 4.294.967.295 = 4$ Gb na memória RAM do computador, que na prática pode ser considerado de 2 Gb. E assim, objetos superiores a 2 Gb não poderão ser trabalhados diretamente na memória RAM do computador. Pode parecer muito, mas 2 Gb podem ser rapidamente consumidos em operações relativamente simples, conforme o tamanho da base de dados original. Isto é uma realidade, sobretudo em análises espaciais, em que além da característica (atributos), coordenadas também devem ser armazenadas. Após inicializado, você já pode adicionar seu primeiro comando:

```
9 + 2
## 11
```

O comando acima executa uma simples soma de 9 e 2, o comando é precedido prompt > e o resultado é apresentado na linha seguinte, precedido do número 1 entre colchetes. O número 1 representa o índice do resultado, ou seja, a posição do elemento no vetor. Os números índices e colchetes apresentados são meramente ilustrativos, o devido tratamento a índices e indexação será dado no capítulo sobre vetores e demais estruturas de dados do R. Na tela do *software*, mais em destaque o console (ou R console) será apresentado algo semelhante à Figura 3.1.

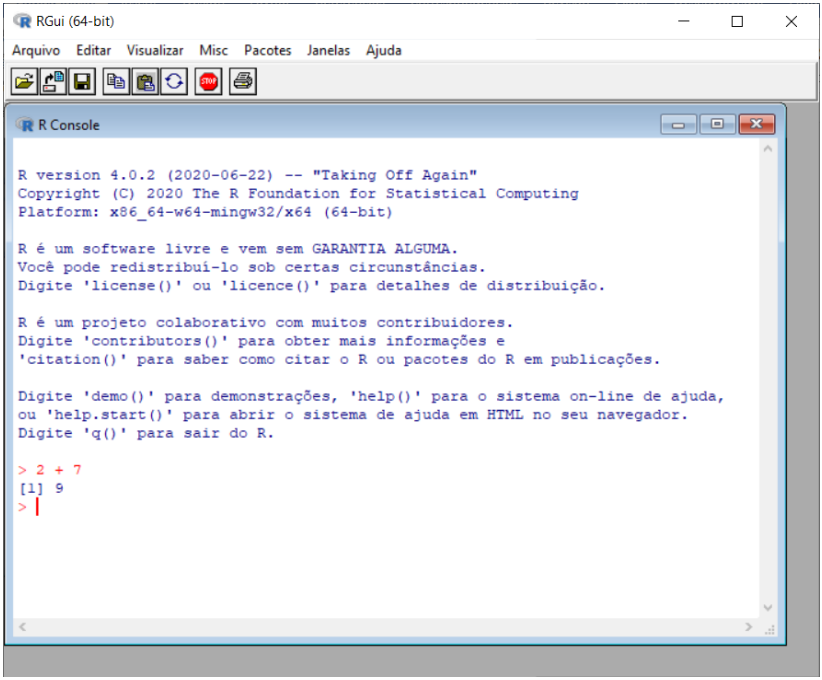

Figura 3.1: Ambiente de trabalho do R.

O console é a parte do software que efetivamente executa as operações, alguns usuários trabalham diretamene nele, em um processo de formular -> digitar -> executar os procedimentos. Porém, a medita que esses procedimentos adquirem o mínimo de complexidade, trabalhar diretamente no console torna-se improdutivo. Então com o auxílio de um editor de textos (como o blocos de notas ou o próprio editor do R, acessado via Arquivo - Abrir script), o usuário trabalha em um processo cíclico de formular digitar, e então o procedimento é executado (enviado/copiado para o console) após a finalização parcial ou final deste ciclo.

Apesar do R disponibilizar um editor de texto específico para editar scripts trata-se de um editor bastante limitado, e assim a ampla maioria dos usuários optam por instalar um segundo software para auxiliar na tarefa de edição de scripts. No caso da linguagem de programação R a maioria esmagadora dos usuários optam pelo RStudio, que pode ser considerado um Ambiente de Desenvolvimento Integrado - *Integrated Development Environment* (IDE).

O RStudio apresenta caracteristicas apreciáveis, tais como, plataforma madura, amplamente utilizada e com um excelente editor de texto, que conta com uma série de funcionalidades, como identificação de erros de sintaxe; complemento de funções e objetos; coloração diferenciada de objetos e estruturas de controle; atalhos de teclado úteis; além de outras funcionalidades, como janelas específicas para plotar figuras, acessar arquivos e bases de dados, consultar os documentos de ajuda das funções Figura 3.2.

O download do RStudio pode ser feito no site https://rstudio.

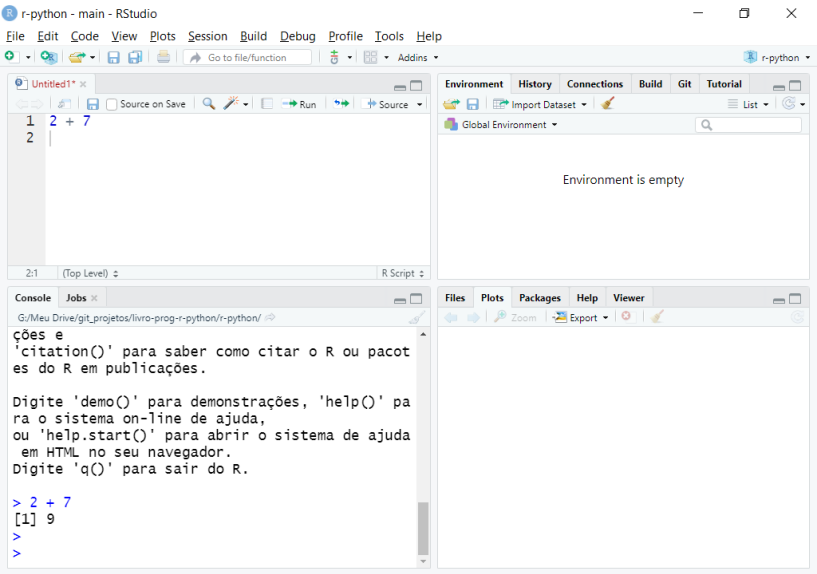

Figura 3.2: Ambiente de trabalho do R acessado via RStudio.

com/products/rstudio/download/. O RStudio não substitui o R, ele é apenas uma interface mais agradável e produtiva para acessar todos os procedimentos poderosos que o R oferece, então antes de instala-lo deve-se instalar o R.

3.2 Operações básicas e execução de funções

A seguir é a apresentado mais algumas operações matemáticas básicas, a partir de agora com a omissão do prompt e adição de comentários nos resultados retornados pelo programa[2].

[2]Essas medidas serão particularmente úteis para facilitar operações de copiar-colar códigos da versão digital deste documento.

```
2 + 3*4 # prioridade da operação multiplicação
## [1] 14
3/2 + 1 # prioridade da operação divisão
## [1] 2.5
3 / (2 + 1) # parênteses estabelecem prioridade
## [1] 1
2 * 3 ^ 2 # potências são indicadas por ^ ou **
## [1] 18
```

Determinados operadores apresentam prioridades sobre os demais, assim como qualquer em qualquer calculadora ou planilha eletrônica. Espaços entre os operados podem ser reservados ou não entre os números, a legibilidade dita a quantidade de espaços reservados. Linhas que apresentam o símbolo **#** definem um comentário e são ignoradas pelo R.

No R todas as funções têm a seguinte forma:

função(argumento(s)obrigatório(s), argumento(s)opcional(is))

Sendo que os argumentos opcionais podem ter um valor padrão pré-estabelecido ou não. Os argumentos estarão sempre entre parênteses sendo separados por vírgula.

```
log(2) # ln - Logaritmo de 2 na base e
## [1] 0.6931472
log(2, 10) # Logaritmo de 2 na base 10
## [1] 0.30103
```

Para descobrir quais são os argumentos da função, bem como o que é retorna você pode pedir ajuda para o R. Para pedir ajuda

ao R e ter acesso à documentação de determinada função, por exemplo, para a função log, digite uma das opções:

```
help(log)
?log
```

Pesquisamos a documentação oficial do programa com muita frequência, praticamente o tempo todo, pelo menos para saber quais são os argumentos necessários para executar determinada função. As opções acima é para situações em que você já sabe o nome do comando, porém sempre pode utilizar o mecanismo de completar funções do RStudio ou do próprio console do R para identificar nomes de funções.

Para pesquisar sobre temas em particular utilize os comandos abaixo. Porém na maioria das vezes é mais rápido e fácil fazer essa pesquisa mais abrangente no Google.

```
??logarithms
help.search("logarithms")
```

Outra função útil de ajuda no R é a função `args`, que lista os argumentos necessários para executar a função de interesse, porém a maioria dos usuários prefere acessar a documentação completa da função, que além de listar, também descreve os argumentos e cada um dos elementos que compõe a função.

```
args(log)
## function (x, base = exp(1))
##   NULL
```

Agora que você já sabe como pedir ajuda no R, vamos continuar com a execução de funções. Se você deixar o primeiro argumento (obrigatório) em branco, vai receber uma mensagem de erro:

```
log( , 3)
## Error: argument "x" is missing, with no default
```

Se o nome dos argumentos estiverem nomeados não é necessário ordena-los de acordo com a especifícação da função.

```
log(x = 2, base = 10) # ordem original
## [1] 0.30103
log(base = 10, x = 2) # ordem inversa
## [1] 0.30103
```

Provavelmente não será necessário alterar a ordem de especificação dos argumentos, como exemplificado acima, inclusive essa alteração é desencorajada. Mas em algumas situações, atribuir argumentos pelo nome é essencial, como no exemplo a seguir, em que o 1º e o 3º argumentos devem ser atribuidos e o segundo deve ser mantido como o padrão. A função `rnorm`[3] é aplicada, ela serve para gerar números aleatórios conforme a função densidade de probabilidade normal. O argumento `n` (número de elementos) é igual a 2 e o desvio padrão (`sd`) é igual igual a 5 (diferente do padrão 1). Observe que o padrão para `mean` (média) é mantido, isto é, continua igual a zero. Se o nome do argumento `sd` não for estabelecido, a função iria executar da mesma forma, porém o número 5 iria ser atribuido para o argumento de `mean`.

[3]A função `rnorm` gera números pseudoaleatórios, e se a semente inicial de geração de números aleatórios não for especificada, o função irá retornar números diferentes em cada execução.

```
args(rnorm)
## function (n, mean = 0, sd = 1)
## NULL
rnorm(2, sd = 5)
## [1] -3.857648  1.359604
```

A seguir é apresentado uma lista de algumas funções básicas e operadores aritméticos:

Função	Descrição
sqrt( )	raiz quadrada
abs( )	valor absoluto
exp( )	exponencial de base "e"
log10( )	logaritmo na base 10
log( )	logaritmo na base "e" ou LN
sin() cos() tan()	funções trigonométricas
asin() acos( ) atan( )	funções trigonométricas inversas
+ - * /	adição, subtração, multiplicação e divisão
ou ^	potência

3.3 Objetos no R: gerar e remover objetos

As entidades nas quais R opera são tecnicamente conhecidas como objetos. Exemplos são vetores de valores numéricos (reais) ou complexos, vetores de valores lógicos e vetores de cadeias de caracteres.

O R é uma linguagem orientada a objetos. Um objeto para o R

significa tanto um banco de dados, como uma tabela, variáveis, vetores, matrizes, funções, etc., armazenados na memória ativa do computador. Para criar um objeto qualquer no R, você deverá sempre usar o operador de atribuição `<-`, gerado pela digitação do sinal de menor e menos.

```
x <- sqrt(9)
```

Pronuncia-se o comando dizendo: o objeto recebe certo valor. Por exemplo, `x <- sqrt(9)`, leia-se, "x recebe a raiz quadrada de 9".O objeto `x`, armazenou a raiz quadrada de 9. Verifique, digitando `x`:

```
x
## 3
```

Existem várias formas de fazer atribuições de objetos além do operador `<-`. Outras três delas são apresentas abaixo.

```
sqrt(9) -> x; x = sqrt(9); assign("x", sqrt(9))
```

Porém convencionalmente os usuários do R adotam o `<-`, então é altamente recomendado que este deve ser adotado, pois com o costume você se sentirá mais confortável com códigos de terceiros e o oposto também é verdadeiro.

Ao se fazer uma atribuição deve-se atentar para o fato de que um objeto substitui o outro de mesmo nome, e nenhuma mensagem de advertência é emitida. Para avaliar o conteúdo do objeto, isto é, imprimir o conteúdo na tela, basta digitar o nome do objeto e pressionar a tecla Enter. Basicamente, esse comando chama internamente a função de impressão `print`.

```
obj1 <- 25
print(obj1)
## [1] 25
obj1
## [1] 25
obj1 <- sqrt(9)
obj1
## [1] 3
```

Na maioria das vezes utilizamos a forma resumida do comando `print`, mas em algumas situações seu uso é obrigatório, como dentro de um processo repetitivo, que será apresentado mais adiante neste livro.

Outra função útil de impressão de objetos é a função `cat` (concatenar e imprimir). Ela serve para concatenar (juntar) objetos e imprimi-los na tela ou até mesmo salva-los em um arquivo, isso mesmo, a impressão dos objetos pode ser direcionada para um arquivo texto. Essa função é muito utilizada para concatenar um texto com resultados derivados da execução de algoritmos, armazenados em objetos. A `cat` é mais flexível e "personalizável" do que a função `print`.

```
cat("O valor do objeto 1 (obj1) é", obj1)
## O valor do objeto 1 (obj1) é 22
```

O comando `print` combinado com a função `paste` pode gerar resultado semelhantes à função `cat`. A função `paste` serve para concatenar (juntar) vetores após converte-los em caracteres (texto).

```
print(paste("O valor do objeto 1 (obj1) é", obj1))
## [1] "O valor do objeto 1 (obj1) é 22"
```

O R reconhece letras maiúsculas e minúsculas como caracteres diferentes, assim como a ampla maioria das linguagens de programação, essa característica recebe o nome de *case sensitive*. Observe o comportamento dos objetos abaixo.

```
a <- 1; A <- 5
nome <- "Eduardo"
Nome <- "outro nome"
a; A; nome; Nome
## [1] 1
## [1] 5
## [1] "Eduardo"
## [1] "outro nome"
```

Durante uma sessão do R, os objetos são criados e guardados por nomes. Para saber quais objetos estão guardados na memória pelo R basta avaliar a aba de Environment do RStudio ou utilizar um dos comandos:

```
objects()
## [1] "a" "A" "nome" "Nome" "obj1" "x"
ls()
## [1] "a" "A" "nome" "Nome" "obj1" "x"
```

Para eliminar um ou mais objetos basta utilizar a função `rm` de remover.

```
rm(x, obj1)
ls()
## [1] "a" "A" "nome" "Nome"
```

Para eliminar todos os objetos pode-se utilizar o comando o comando abaixo, porém a maioria dos usuários preferem utilizar a opção disponível no menu do programa para essa funcionalidade, em Misc – Remover todos os objetos, no R padrão, e em Session – Clear Workspace, no RStudio.

```
rm(list = ls(all = TRUE))
```

Se o seu interesse é apenas limpar o console, pode-se utilizar o atalho ctrl+L, porém esse atalho não remove os objetos.

É comum errar alguns comandos quando se está trabalhando com o R. Seja pela falta de familiaridade com o comando ou então por algum erro de digitação. Para evitar ter que escrever o comando todo de novo, utilize a seta do teclado de direção para cima para pesquisar todo histórico de comandos utilizados na sessão atual. Quando encontrar o comando desejado faça as devidas correções e execute novamente. Porém é altamente recomendado que o processo de criação de um procedimento seja feito utilizando o editor de texto, assim a preocupação de recuperar comandos é eliminada, uma vez todos os comandos estarão salvos no arquivo texto de edição. E no caso de um eventual erro, basta fazer a correção no arquivo texto e executar o procedimento novamente no console.

Capítulo 4

CRIAÇÃO DE FUNÇÕES

No tópico anterior foi mostrado como executar uma função no R, neste tópico iremos mais além, será apresentado os componentes básicos para você criar suas próprias funções e se beneficiar de todas as suas vantagens.

Uma das maiores potencialidades do R é que permite ao usuário definir suas próprias funções de forma simples e fácil. Isso o torna uma ferramenta poderosa para criar e testar novas metodologias. As funções são utilizadas para praticamente tudo e inclusive para criar novas funções. No R as funções apresentam papel de destaque, pois é a principal forma de interagir com as rotinas nativas da linguagem.

4.1 Motivação para criar funções

De uma visão mais prática as funções são úteis por uma série de fatores, podemos citar **estender/expandir as funcionali-**

dades de um sistema base, isto é, criar novos procedimentos para o sistema; serve para **encapsular parte do código** que executa a mesma funcionalidade ao longo do processo, e assim organizar melhor o código, além de tornar o processo mais seguro, uma vez que a execução da função ocorre em ambiente local (diferente do ambiente global da sessão), isso pode evitar efeitos indesejáveis, os processos do ambiente local não afeta o global e também não é afetado; permite **reutilizar códigos**, e assim evitar repetição desnecessária de códigos, isso facilita muito o processo de criação e manutenção do código, além de evitar o problema de copiar e colar: as funções são extremamente úteis para reduzir a duplicação de códigos [BOX1].

BOX1 - Métrica do código repetido: Existem softwares, tais como CodeClimate e SonarQube, que avaliam a qualidade do código desenvolvido frente a diferentes perspectivas, uma delas é a quantidade de códigos repetidos. Códigos repetidos são nocivos para o programa, uma vez que o torna mais extenso e complexo. Em caso de necessidade de alteração no código repetido a alteração deverá ser feita em todos as repetições, sem exceções, e isso pode ser trabalhoso e com alta susceptibilidade à erros. O ideal seria encapsular este trecho de código repetido em uma função.

De uma visão mais geral, o papel primordial das funções é o de **abstração**. Abstração é difícil de ser definida, mas consiste no ato de isolar elementos em detrimento a outros. A abstração é muito útil para simplificar eventos e assim facilitar a resolução de problemas, por exemplo, isola-se apenas aquilo que é útil e relevante, todo resto então deve ser esquecido ou eliminado.

Um exemplo prático de abstração seria o uso de um projetor de slides. Um professor não precisará saber que tipo de lâmpada aquele projetor utiliza, nem mesmo a quantidade de lumens da lâmpada para utilizá-lo em uma aula. Muito provavelmente o conhecimento necessário será o de instalar em um computador e a funcionalidade de liga/desliga. Apenas com esse conhecimento, o professor já é capaz de ministrar sua aula. Observe que um nível de abstração diferente é requerido para um técnico que ganha a vida dando manutenção nesses equipamentos, possivelmente ele vai ter que entender de lâmpadas e lumens.

Na programação, um exemplo poderia ser a função nnet da biblioteca de mesmo nome do R, essa função serve para ajustar redes neurais artificiais à um conjunto de dados. A interface com o usuário é simples e intuitiva, indicada para a maioria das pessoas, que irão utilizar essa função como uma ferramenta, como um Engenheiro Florestal que irá predizer a produtividade de determinada cultura utilizando variáveis climáticas. Um nível de abstração diferente será requerido para um profissional da matemática ou ciência da computação, que será responsável por implementar uma função que executa uma rede neural artificial.

4.2 Elementos básicos de uma função

A nova função R que você construir poderá ser completamente nova (um novo modelo que você está testando, por exemplo) ou apenas uma modificação personalizada de uma função do R já existente. Você pode desejar ainda usar as funções já existentes de modo repetido no seu conjunto de dados, isto facilitará em muito o seu trabalho, já que as tarefas a serem realizadas ficarão

incorporadas em uma única função. A sintaxe[BOX2] básica de uma função do R é apresentada a seguir.

Sintaxe:

```
nome <- function(argumentos) {
    sequências de instruções (corpo)
    return(argumento)
}
```

nome: aquele que você escolhe para dar a função. O nome é um elemento essencial de uma função, para existir a função deverá possuir um nome;

argumentos: lista de expressões a serem usadas dentro da função, que podem ser obrigatórios ou opcionais. Os argumentos também podem apresentar um padrão pré-estabelecido. Argumentos não são essenciais, embora incomum, funções sem argumentos podem ser construída;

corpo: é a parte da função que realmente trabalha, é constituído por expressões R que serão avaliadas sequencialmente quando executadas. Não faz sentido construir uma função sem corpo, isto é, que não executa qualquer instrução, porém é possível construir uma função sem corpo e nenhum erro de sintaxe seria emitido.

return: é o último valor calculado, e pode vir acompanhado para função reservada return ou não. O retorno de uma função não é essencial, podemos ter interesse em apenas imprimir o resultado na tela. O retorno é útil se este resultado será utilizado posteriormente no decorrer no processo, assim o retorno poderá ser armazenado em um objeto após a execução da função.

BOX 2 - Sintaxe, semântica, lógica: A **sintaxe** refere-se às regras que ditam a composição de textos com significado lógico para determinada linguagem programação. Os erros de sintaxe são erros no código e bloqueiam a execução de um processo. Exemplos seria digitar de forma errada a palavra reservada function ou esquecer de fechar um parêntese. O RStudio possui mecanismos para checar alguns erros sintaxe na edição do script, antes mesmo da execução. Os erros de sintaxe geralmente são fáceis de serem detectados. Além dos erros de sintaxe ainda podemos ter os erros de **semântica**, que envolvem códigos tecnicamente corretos, mas apresentam problemas com o significado. Esses erros geralmente só podem ser identificados no momento da execução do código. Um exemplo seria tentar ler um arquivo que não existe no computador. Há também os erros de **lógica**, esses erros são mais difíceis de serem detectados, a sintaxe e semântica estão corretas, mas mesmo assim o código não executa da maneira com que o programador imaginou. Os erros de lógica são críticos porque não podem ser detectados pelo editor ou interpretador, a identificação seria de responsabilidade inteiramente do programador. Exemplo seria usar uma operação errada no processo, como utilizar soma em vez multiplicação. O resultado seria gerado corretamente, porém é de nosso interesse a multiplicação e não a soma.

Podemos criar nossa própria função para dar boas-vindas ao usuário, observa as diferentes versões para a tarefa.

```r
# Versão 1 - sem argumentos
boas_vindas <- function() { # criar a função
    cat("Seja bem vindo!")
}
boas_vindas() # executar a função
## Seja bem vindo!

# Versão 2 - com argumentos
boas_vindas <- function(nome) {
    cat("Seja bem vindo", nome, "!")
}
boas_vindas("Eduardo")
## Seja bem vindo Eduardo !

# Versão 3 - com argumento padrão
boas_vindas <- function(nome = "Aluno") {
    cat("Seja bem vindo", nome, "!")
}
boas_vindas()
## Seja bem vindo Aluno !

# Versão 4 - com argumento padrão e retorno
boas_vindas <- function(nome = "Aluno") {
    frase <- paste("Bom dia", nome, "!")
    return(frase)
}
frase_boas_vindas <- boas_vindas("Eduardo")
frase_boas_vindas # mostrar o objeto resultante
## [1] "Bom dia Eduardo !"
```

O próximo exemplo demostra o uso do resultado retornado pela execução da função em uma análise posterior. Além de retornar o resultado numérico, a função também imprime na tela o resultado combinado com um texto explicativo.

```
lucro <- function(receita, custo) {
    valor_lucro <- receita - custo
    cat("O lucro foi de R$", valor_lucro, "\n")
    return(valor_lucro)
}
lucro_projeto1 <- lucro(6000, 4500)
## O lucro foi de R$ 1500
lucro_projeto2 <- lucro(4000, 5000)
## O lucro foi de R$ -1000
lucro_total <- lucro_projeto1 + lucro_projeto2
lucro_total
## [1] 500
```

A função computa o lucro de dois diferentes projetos, os resultados são somados para computar qual seria o lucro total. Observe que foi incluído o texto `\n` na função de concatenar e imprimir, isso foi necessário para evitar que a impressão e o resultado retornado sejam apresentados na mesma linha. O texto `\n` indica para o programa pular de linha. Vale destacar que se optar por utilizar a função print combinada com a função paste não seria necessário incluir `\n`, pois por padrão a função print já pula para próxima linha após a execução.

4.3 Escopo

Regras de espoco, ou simplesmente escopo, é um conjunto de regras destinado a controlar o acesso de variáveis durante a execução de um processo. Não existe um padrão universal de regras de escopo, mas em geral, não existe grandes variações de regras entre a maioria das linguagens de programação.

No R temos dois escopos básicos de variáveis, o global e o local, o primeiro, em que as variáveis podem ser acessadas em qualquer momento dentro da função, independente se a variá-vel foi criada dentro da função ou não, já o segundo, as variá-veis externar só podem ser acessadas na função se passadas como parâmetros, além disso, as variáveis criadas dentro da função não afeta o ambiente externo (global).

Capítulo 5

OPERAÇÕES CONDICIONAIS

Para você dizer em algum momento que sabe programar será necessário dominar dois tópicos básicos: operações condicionais e operações de repetição. Ambos são operações de controle de fluxo e são consideradas essenciais para programar qualquer procedimento, por mais simples que seja. **As operações condicionais estão relacionadas com tomada de decisão entre diferentes alternativas, isto é, escolha de uma opção entre duas ou mais possibilidades.** Para ser capaz de compreender as operações condicionais é necessário entender pelo menos 4 tópicos: os operadores condicionais, operadores de comparação, operadores lógicos e operadores condicionais aninhados.

5.1 Operadores condicionais

O primeiro tópico a ser entendido de operações condicionais é sobre os **operadores condicionais**, que é a estrutura geral das operações de condição. A maioria das linguagens de programação

apresentam um operador condicional básico, que geralmente é encontrado em três versões básicas, apresentadas na Figura 5.1.

Operadores condicionas	
Se Condições: Tarefas	**[se]** **[if]**
Se Condições: Tarefas$_1$ **Caso Contrário:** Tarefas$_2$	**[se] [caso contrário]** **[if] [else]**
Se Condições$_1$: Tarefas$_1$ **Caso Contrário Se Condições$_2$:** Tarefas$_2$ **Caso Contrário:** Tarefas$_3$	**[se] [caso contrário se] [caso contrário]** **[if] [else if] [else]**

Figura 5.1: Operadores condicionais.

As decisões nos operadores condicionais são controladas pelas condições, as condições são construidas utilizando uma ou mais expressões que devem ser avaliadas em relação a sua veracidade. Se a condição for verdadeira, uma sequência de instruções (tarefas) deverão ser executadas, caso for falsa, isto é, caso contrário, uma sequência alternativa de instruções deverá ser executada.

5.2 Operadores de comparação

As expressões de condições geralmente são construidas utilizando os **operadores de comparação** (Figura 5.2) e combinadas utilizando os **operadores lógicos** no caso de mais de uma expressão.

Operadores de comparação	
==	Igual
!=	Diferente
>	Maior que
>=	Maior que ou igual a
<	Menor que
<=	Menor que ou igual a

Figura 5.2: Operadores de comparação.

Na sequência é apresentado o operador condicional `if`. Na Figura 5.3 é apresentado o fluxograma de ações referentes à decisão.

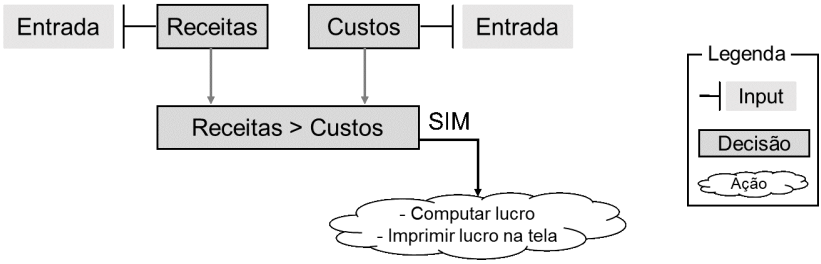

Figura 5.3: Decisão e ações em relação à avaliação de projetos para o exemplo if.

Observe que se a condição for falsa, isto é, se receitas forem menor ou igual aos custos, o código não executa instrução nenhuma, nem imprime na tela e nem computa o lucro (prejuízo no segundo exemplo).

```
# Operador condicional if
receitas <- 7500
custos <- 5000
```

```
if(receitas > custos){ # imprime o lucro na tela
   lucro <- receitas - custos
   cat("O lucro foi de R$", lucro)
}
## O lucro foi de R$ 2500
receitas <- 4500
custos <- 5000
if(receitas > custos){ # nenhuma tarefa é executada
   lucro <- receitas - custos
   cat("O lucro foi de R$", lucro)
}
##
```

O código será aperfeiçoado incluindo a instrução else (caso contrário) e as tarefas relacionadas a ela. Outro aperfeiçoamento será o encapsulamento do código em uma função, isso para evitar a repetição de código como apresentado acima. O fluxograma é apresentado na Figura 5.4 e o código é apresentado a seguir.

O fluxograma da Figura 5.4 apresenta uma sequência de instruções para o caso do projeto operar no prejuízo (Receitas não são maiores que custos), além disso o retorno da função também é representado no fluxograma.

```
# Operador condicional if-else
aval_econ <- function(receitas, custos) {
   if (receitas > custos) {
      lucro <- receitas - custos
      cat("O lucro foi de R$", lucro, "\n")
   } else {
```

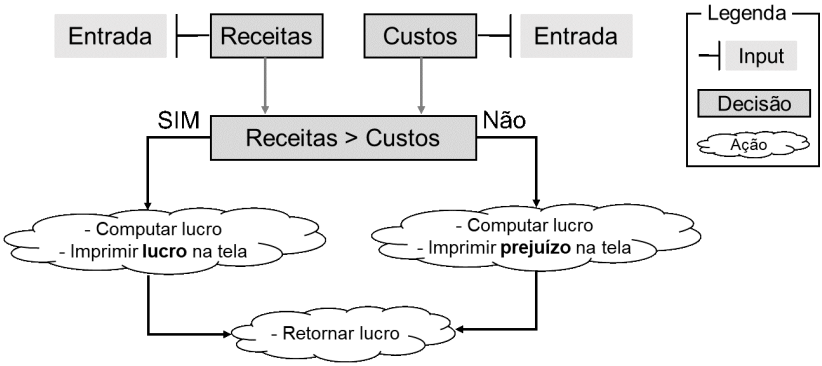

Figura 5.4: Decisão e ações em relação à avaliação de projetos para o exemplo if-else.

```
      lucro <- receitas - custos
      cat("O prejuízo foi de R$", abs(lucro), "\n")
   }
   return(lucro)
}
aval_econ(7500, 5000)
## O lucro foi de R$ 2500
## [1] 2500
aval_econ(4500, 5000)
## O prejuizo foi de R$ 500
## [1] -500
```

Observe que no bloco `else`, foi incluido a função `abs` no objeto de lucro, a função `abs` retornar o valor absoluto de um número, isto é, se o valor é negativo passa a ser positivo. Isso foi feito porque no texto explicativo já acompanha a palavra prejuízo, e assim já indicando ser um valor negativo.

É possivel apresentar mais um aperfeiçoamento, observe que a avaliação econômica de um projeto ainda pode apresentar um terceiro cenário, além dos dois já apresentados, de lucro e prejuízo. Seria o cenário de nem lucro e nem prejuízo, isto é, elas por elas, receitas iguais aos custos. O fluxograma é apresentado na Figura 5.5.

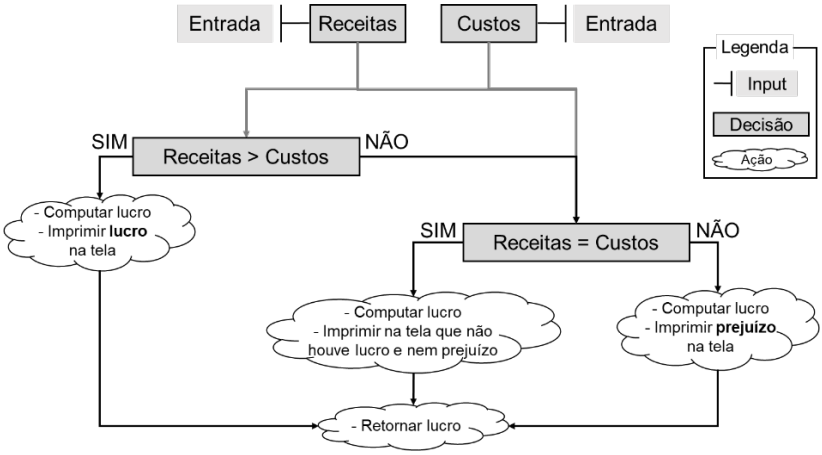

Figura 5.5: Decisão e ações em relação à avaliação de projetos para o exemplo if-else if-else.

Os operadores condicionais funcionam realmente como um processo de tomada de decisão, se a resposta para primeira pergunta (Receitas são maiores do que Custos?) for sim, as tarefas são executadas e o processo caminha para seu encerramento, isto é, para o retorno do resultado (valor de lucro). Observe que não será necessário fazer a segunda pergunta (Receitas são iguais aos Custos?) referente à segunda decisão.

```
# Operador condicional if-else if-else
aval_econ <- function(receitas, custos) {
   if (receitas > custos) {
      valor <- receitas - custos
      cat("O lucro foi de R$", valor, "\n")
   } else if (receitas == custos) {
      valor <- receitas - custos
      cat("Não apresentou lucro e nem prejuízo \n")
   } else {
      valor <- receitas - custos
      cat("O prejuizo foi de R$", abs(valor), "\n")
   }
   return(valor)
}
```

A aplicação da função `aval_econ` para três projetos distíntos é apresentada a seguir, e por fim, o somatório dos resultados obtidos é computado para identificar o lucro total. Observe que as avaliações de cada projeto são armazenadas em objetos para posterior utilização, isso só é possível devido ao retorno do resultado pela função. Apenas com a aplicação da função `print` não seria possível calculos posteriores.

```
proj1 <- aval_econ(7000, 9000)
O prejuizo foi de R$ 2000
## proj1
## [1] -2000
proj2 <- aval_econ(10000, 6500)
## O lucro foi de R$ 3500
proj2
```

```
## [1] 3500
proj3 <- aval_econ(5000, 5000)
## Não apresentou lucro e nem prejuízo
proj3
## [1] 0
lucro_total <- proj1 + proj2 + proj3
lucro_total
## [1] 1500
```

5.3 Operadores lógicos

Os operadores lógicos principais são o `ou` e `e` (Figura 5.6), representados pelos símbolos `|` e `&`, respectivamente. O operador `&` pode ser considerado mais restritivo, uma vez que só retorna verdadeiro se todas as expressões de condição forem verdadeiras. Já o operador `|` é considerado mais permissivo, pois basta uma expressão ser verdadeira para toda a condição retornar verdadeiro no final. A escolha do operador, obviamente vai depender do problema em questão. Ainda existe a negação lógica, representada pelo símbolo `!`, que inverte o resultado de uma expressão condicional, por exemplo, se o resultado for verdadeiro, é então convertido em falso e vice-versa. Também existe o operador `xor` (ou exclusivo), que retorna verdadeiro se ambos os valores de entrada forem diferentes entre si, e retorna falso se forem iguais.

O exemplo a seguir demonstra o uso dos operadores lógicos, o objetivo é identificar qual será a quantidade de área plantada no ano seguinte baseando-se na performance do ano atual. Se a relação receita / custo do projeto for menor um, isto é, obteção

Operadores lógicos			
e "&"	ou "|"	ou exlusivo "xour"	Negação "!"
V e V = V	V ou V = V	xor(V, V) = F	!V = F
V e F = F	V ou F = V	xor(V, F) = V	!F = V
F e V = F	F ou V = V	xor(F, V) = V	
F e F = F	F ou F = F	xor(F, F) = F	

Figura 5.6: Os operadores lógicos

de prejuízo, não será feita expansão de plantio, deve-se plantar exatamente a quantidade de área plantada do ano atual; se a relação receita / custo for entre 1 e 1,5, será feita expansão de plantio de 20%; se a relação for entre 1,5 e 2, a expansão será de 50%; mas caso a relação for superior a 2, a expansão será de 100% e a área de plantio será dobrada.

```
area_plantio <- function(receitas, custos, area) {
   prop <- receitas / custos # proporção receitas / custos
   if (prop <= 1) {
      area_seg <- area # aumento de 0%
      cat(paste("Proporção Receita/Custo:", round(prop, 1),
                "; Área de plantio:", area_seg, "ha\n"))
   } else if (prop > 1 & prop <= 1.5) {
      area_seg <- area * 1.2 # aumento de 20%
      cat(paste("Proporção Receita/Custo:", round(prop, 1),
                "; Área de plantio:", area_seg, "ha\n"))
   } else if (prop > 1.5 & prop <= 2) {
      area_seg <- area * 1.5 # aumento de 50%
      cat(paste("Proporção Receita/Custo:", round(prop, 1),
                "; Área de plantio:", area_seg, "ha\n"))
```

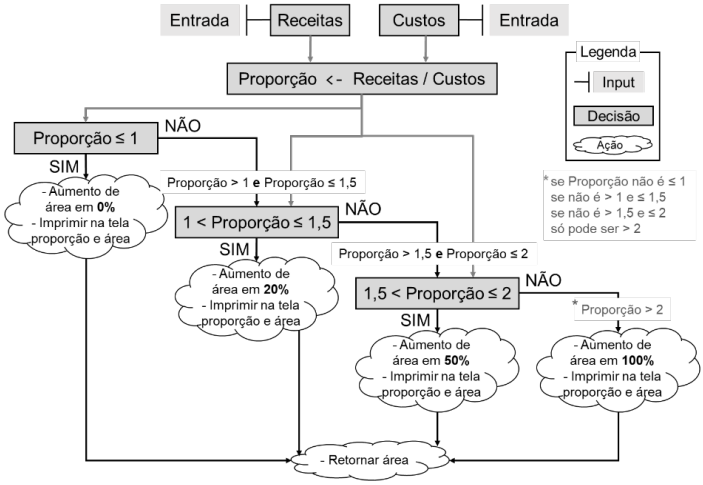

Figura 5.7: Decisão e ações em relação à quantidade de área de plantio para o exemplo if-else if-else, com auxílio de operadores lógicos.

```
    } else {
        area_seg <- area * 2 # aumento de 100%
        cat(paste("Proporção Receita/Custo:", round(prop, 1),
                  "; Área de plantio:", area_seg, "ha\n"))
    }
    return(area_seg)
}
```

A aplicação da função `area_plantio` é apresentada a seguir.

```
receitas <- 5000 # R$
custos <- 5500 # R$
area_atual <- 200 # ha
area_plantio(receitas, custos, area_atual)
```

```
## Proporção Receita/Custo: 0.9 ; Área de plantio: 200 ha
## [1] 200
custos <- 3600 # R$
area_plantio(receitas, custos, area_atual)
## Proporção Receita/Custo: 1.4 ; Área de plantio: 240 ha
## [1] 240
area_plantio(receitas, 2800, area_atual)
## Proporção Receita/Custo: 1.8 ; Área de plantio: 300 ha
## [1] 300
area_plantio(receitas, 2400, area_atual)
## Proporção Receita/Custo: 2.1 ; Área de plantio: 400 ha
## [1] 400
```

5.4 Operadores condicionais aninhados

Os operadores lógicos podem ser substituídos por *operadores condicionais aninhados** (ou encaixados). Os operadores condicionais aninhados, são basicamente, um operador condicional dentro do outro, e as vezes vale a pena recorrer para essas estruturas de condição por questões de simplificação. Com os condicionais aninhados o código pode ficar mais simples de implementar e entender.

O próximo exemplo demostra o uso dos condicionais aninhados. O fluxograma em Figura 5.8 será implementado utilizando a linguagem de programação R.

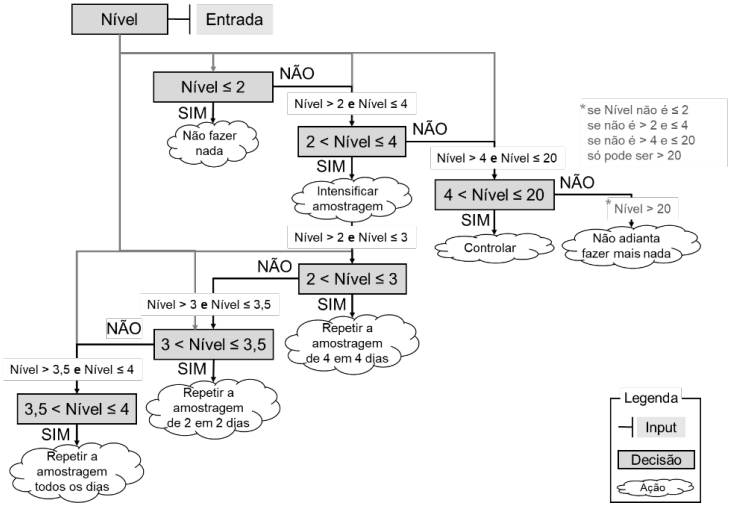

Figura 5.8: Decisões em relação ao nível de infestação para controle de pragas.

```
controle_pragas <- function(nivel) {
   if (nivel <= 2) { # (1) esperar
      cat("Sem danos econômicos: NÃO TENTAR CONTROLAR.\n")
      cat("Manter programação normal de controle de pragas.\n")
   } else if (nivel > 2 & nivel <= 4) { # (2) intensificar amos
      cat("Sem danos econômicos: NÃO TENTAR CONTROLAR.\n")
      cat("Mas intensificar a amostragem.\n")
      if (nivel > 2 & nivel <= 3) { # amostragem opção 1
         cat("Repetir a amostragem de 4 em 4 dias.\n")
      } else if (nivel > 3 & nivel <= 3.5) { # amostragem opção
         cat("Repetir a amostragem de 2 em 2 dias.\n")
      } else if (nivel > 3.5 & nivel <= 4) { # amostragem opção
         cat("Repetir a amostragem todos os dias.\n")
      }
```

```
   } else if (nivel > 4 & nivel <= 20) { # (3) ação
      cat("Ação!!! TENTAR CONTROLAR.\n")
   } else { # (4) já era
      cat("Não adianta fazer nada: NÃO TENTAR CONTROLAR.\n")
      cat("Os dados econômicos já são elevados demais.\n")
   }
}
```

A aplicação da função `controle_pragas` é apresentada a seguir.

```
controle_pragas(1.75)
## Sem danos econômicos: NÃO TENTAR CONTROLAR.
## Manter programação normal de controle de pragas.
controle_pragas(3.1)
## Sem danos econômicos: NÃO TENTAR CONTROLAR.
## Mas intensificar a amostragem.
## Repetir a amostragem de 2 em 2 dias.
controle_pragas(4.6)
## Ação!!! TENTAR CONTROLAR.
controle_pragas(21.2)
## Não adianta fazer nada: NÃO TENTAR CONTROLAR.
## Os dados econômicos já são elevados demais.
```

Capítulo 6

OPERAÇÕES DE REPETIÇÃO (LOOP)

As operações de repetição (**loop**) são a essência da computação em geral, pois foi justamente da necessidade de executar tarefas repetidas vezes (e de forma rápida) é que impulsionou o desenvolvimento e popularização dos computadores. Além disso, muitas dessas tarefas são maçantes e complexas, o que reforça a importância dessa forma de controle de fluxo.

O **loop** permite executar trechos do algoritmo quantas vezes for necessária, a quantidade de vezes pode ser defina a priori pelo programador, mas geralmente é determinada de acordo com o problema em questão. Cada repetição pode ser chamada de iteração[BOX3].

BOX3 - Iteração versus interação: Iteração é diferente de interação! Iteração é um processo repetitivo, já interação diz respeito à comunicação em dois sentidos, isto é, que interage, como algum tipo de interface gráfica de software que interage

com o usuário.

Geralmente, a quantidade de iterações é controlada com o auxílio de uma variável específica, que a cada iteração é incrementada, isto é, somar uma quantidade (geralmente unitária; igual a 1) ao valor atual dessa variável (`i <- i + 1`). Decremenos (`i <- i - 1`) também podem ser necessários, porém com menor frequência. Essa variável especial, que é incrementada ou decrementada, também pode ser entendida como um contador (contador de iterações).

É possível executar processos repetitivos via programação sequencial, para isso, basta copiar (repetidamente) o trecho do algoritmo que requer repetição, porém essa estratégia pode ser impraticável conforme a quantidade de repetições. Além disso, com as cópias, o código aumenta de tamanho e complexidade, tornando o algoritmo difícil de ser entendido, corrigido, adaptado e mantido.

6.1 Loop: visão geral

Existem dois tipos básicos de loop, os condicionais e os incondicionais. Os condicionais podem ser subdivididos em duas partes: em que a condição é avaliada inicialmente (**enquanto - while**) e em que é avaliada posteriormente (**repita - repeat**). Já os incondicionais, as instruções são repetidas até que toda uma sequência seja avaliada (para toda sequência - for). Uma visão geral é apresentada no Figura 6.1.

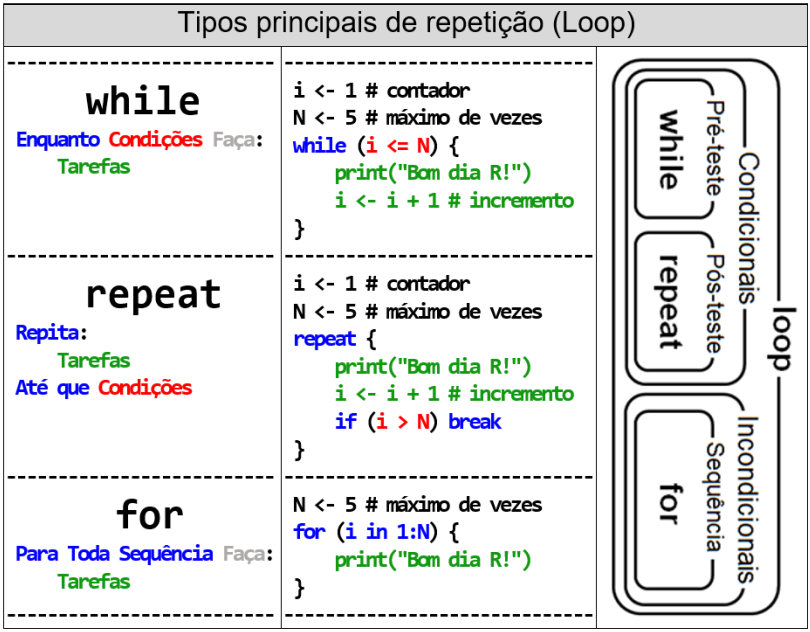

Figura 6.1: Visão geral dos processos de repetição em loop.

A Figura acima apresenta as três formas de implementar um loop no R, utilizando as estruturas de controle: `while`, `repeat` e `for`. É importante dizer que é possível desenvolver qualquer algoritmo dominando apenas uma dessas formas, porém conforme o problema uma pode ser mais indicada (por ser mais lógico, natural, mais fácil de entender) em relação a outra.

O loop condicional com pós-teste é encontrado em muitas linguagens de progamação com o nome `do-while`, como C/C++ e Java, no caso do R, essa estrutura de repetição é representada pelo loop `repeat`. Esse loop requer a implementação explicita do teste condicional (utilizando a estrutura de controle condicional `if`), além disso é necessário a utilização da estrutura de controle auxiliar `break`, afim de interromper o processo repetitivo do loop quando determinada condição for atendida. Devido a este fato, no `repeat` do R, o teste condicional para continuidade de repetição pode ser aplicado em qualquer momento do loop (início, meio ou fim). O `next` é uma estrutura de controle auxiliar, assim como `break`, porém ao invés de encerrar o loop definitivamente, o `next` encerra apenas a iteração atual.

6.1.1 Repetição com copiar-colar

O exemplo a seguir tem como objetivo imprimir na tela o texto "Bom dia R!" repetidas vezes. Vamos supor que estamos de excelente humor e assim o interesse é imprimir na tela a mensagem de bom dia 4 vezes. Inicialmente vamos adotar uma solução ingênua, utlizando a estratégia de copiar-colar para realizar a tarefa. O número da repetição é adicionada inicialmente apenas para conferencia da quantidade.

```
cat("1 - Bom dia R!\n")
## 1 - Bom dia R!
cat("2 - Bom dia R!\n")
## 2 - Bom dia R!
cat("3 - Bom dia R!\n")
## 3 - Bom dia R!
cat("4 - Bom dia R!\n")
## 4 - Bom dia R!
```

6.1.2 Repetição com o loop `while`

Já a próxima solução será baseado no loop `while`, irei começar por ele, pois talvez seja o mais intuitivo e fácil de entender. A essencia do while é: enquanto uma condição for verdadeira a tarefa deverá ser repetida.

```
i <- 1 # contator
N <- 4 # quantidade de vezes
while (i <= N) { # enquanto condição
    cat(i, "- Bom dia R!\n" ) # tarefa
    i <- i + 1 # incremento
}
## 1 - Bom dia R!
## 2 - Bom dia R!
## 3 - Bom dia R!
## 4 - Bom dia R!
```

Para este exemplo especificamente, o loop `while` necessita da variável adicional de contador, que deve ser inicializada antes

do loop e incrementada ao final de cada iteração, essa variável é responsável pelo controle da quantidade de vezes que a tarefa deve ser executada.

6.1.3 Repetição com o loop `repeat`

A solução com o loop `repeat` é apresentada a seguir.

```
i <- 1 # contator
N <- 4 # quantidade de vezes
repeat { # repetir
    cat(i, "- Bom dia R!\n" ) # tarefa
    i <- i + 1
    if (i > N) break
}
## 1 - Bom dia R!
## 2 - Bom dia R!
## 3 - Bom dia R!
## 4 - Bom dia R!
```

A diferença básica do `repeat` e o `while` é que a condição é avaliada no while a condição é avaliada no início do loop, já no `repeat` a condição geralmente é avaliada no final. Na prática, o loop `while` é geralmente mais utilizado do que o loop `repeat`, mas o `repeat` é particularmente útil quando a tarefa prevista deve ser executada pelo menos uma vez, e isso é garantido pelo `repeat`, já que a avaliação da condição só é feita no final da iteração.

No nosso exemplo de imprimir a frase de bom dia, o loop `repeat` seria a solução socialmente mais adequada, pois independente do

do nível de humor, pelo menos uma saudação de bom dia será feita. Para comprovar, teste o código do `repeat` com a quantidade de vezes `N` igual a 0. Um exemplo mais prático e útil seria uma função que aplica um filtro para correção de valores de pixel de uma imagem de satélite. A aplicação do filtro deve ser repetida enquanto houver pixels com valores problemáticos, e vale a pena destacar que se o usuário executou a função, claramente ele considera que a aplicação do filtro deve ser feita pelo menos uma vez. Essa solução poderia ser feita com o `while`, mas a solução com o `repeat` é mais lógica e natural e para alguns problemas pode ser mais econômico em termos de quantidade de código. A solução para este problema pode ser encontrada na seção de Exemplos Extras do livro.

6.1.4 Repetição com o loop `for`

Dentre os três tipos de loop, o `for` é o mais utilizado. A componente essencial do loop `for` é a sequência a ser utilizada. É importante destacar que a sequência do loop for não necessariamente são valores sequências, como 1, 2, 3, 4, 5, sequência no `for` seria uma coleção de elementos que podem ser acessados por sua posição (ou índice). A origem do termo sequência está relacionado com a maneira em que os elementos são armazenados na memória do computador.

Vetor em um loop `for`

No R uma sequência pode ser representada por um vetor[1]. Um vetor é uma coleção de elementos de apenas uma dimensão e composta de um único tipo de elemento, tal como números inteiros (interger), números reais (numeric), lógicos (logical), texto (character).

No R não existe conceito de escalar, assim: `a <- 123.4` é um vetor de apenas um elemento (vetor unitário) e `x <- c(2, 5, 18, 3)` é um vetor de 4 elementos. A função `c()` serve para combinar/juntar elementos, após sua execução, retorna um vetor de comprimento/tamanho (`length`) correspondente à quantidade de elementos combinados.

Existem diversas maneiras de gerar vetores no R, uma das principais é utilizando a função c(), também é possível gerar vetores de elementos sequenciais, isto é, uma coleção de elementos com valores realmente sequências. Por exemplo, o operador `:` pode ser utilizado para gerar sequências, e assim `1:30` gera um vetor `c(1, 2, ..., 29, 30)`. Também é possível construir sequências decrescentes 30:1 com o operador `:`. A função `seq` é a função do R mais geral para criar sequências, e os principais argumentos dessa função são: `from = valor_inicial`, `to = valor_final`, `by = incremento` e `length = comprimento_da_sequência`. O comando `seq(from = 1, to = 30, by = 1)` gera o mesmo resultado do operador `:` para `1:30`.

[1]Neste momento, uma descrição sucinta de vetor será feita. No capítulo sobre estrutura de dados, uma descrição mais completa será apresentada, também serão apresentadas outras estruturas.

Indexação em um vetor no `for`

Uma característica relevante dos vetores é a possibilidade de acessar cada um de seus elementos via indexação, isto é, acessar o elemento de acordo com seu índice, que corresponde a posição do elemento no vetor. Por exemplo: No vetor `x <- c(2, 5, 18, 3)`, para acessar o terceiro elemento, basta utilizar a instrução `x[3]` (nome do vetor - abre colchete - índice/posição do elemento - fecha colchete), e o resultado será o retorno do valor numérico `18`, já o quarto e último elemento, `x[4]`, que irá retornar o valor `3`.

Ao tentar acessar um índice inexistente no vetor `x`, como um quinto elemento (`x[5]`), um `NA` será retornado. `NA` é abreviação de *Not Available* (dados não disponível) e corresponde a um código de valor ausênte/inexistente/perdido. O programador então deve destinar atenção extra, pois nenhuma mensagem de erro será emitida. Em algumas linguagens de programação o acesso inregular de um índice faz com que o processo seja interrompido e uma mensagem de erro é emitida, como no Python. Já em outras linguagens, é simplesmente retornado o valor que estava previamente armazenado na memória naquela posição, o que requer ainda mais atenção do programador no momento de acessar elementos em um vetor.

Os códgios a seguir são utilizados para criar vetores, com exceção do primeiro, os demais são utilizados para criar sequências verdadeiras. Essas sequências são extremamente úteis no loop `for`, pois são utilizadas para acessar elementos de vetores pela posição via indexação. Observe que os vetores `vetor1`, `vetor2` e `vetor3` são exatamente iguais, apesar de serem obtidos de maneiras difer-

entes.

```
vetor <- c(2, 5, 18, 3)
vetor1 <- c(1, 2, 3, 4)
vetor2 <- seq(1, 4, 1)
vetor3 <- 1:4
```

A seguir é apresentado a solução para imprimir na tela a saudação de bom dia utilizando o loop `for`.

```
N <- 4 # quantidade de vezes
for (i in 1:N) { # faça para toda sequência
    cat(i, "- Bom dia R!\n" ) # tarefa
}
## 1 - Bom dia R!
## 2 - Bom dia R!
## 3 - Bom dia R!
## 4 - Bom dia R!
```

Obs: não contador... incremento implícito.