

Estatística descritiva (1ed)
Apostilas de aula com exemplos em R

Djuri Vieira John Fabio Aguilar Sánchez
Luis Francisco Gómez López

2023-12-19

Índice

| | |
|---|-----------|
| Bem-vindos | 1 |
| Prefácio | 3 |
| I Estatística e dados | 5 |
| 1 Visão geral | 7 |
| 2 Dados | 9 |
| II Visualização de dados | 11 |
| 3 Tabelas | 13 |
| 4 Gráficos 2D | 15 |
| III Medidas-resumo | 17 |
| 5 Medidas de tendência central | 19 |
| 6 Medidas de posição | 21 |
| 7 Medidas de dispersão | 23 |
| 8 Medidas de forma | 25 |
| IV Probabilidade | 27 |
| 9 Experimento aleatório e espaço de probabilidade | 29 |
| 10 Interpretações da Probabilidade | 31 |
| 11 Consequências dos axiomas de probabilidade | 33 |
| 12 Independência e probabilidade condicional | 35 |

| | |
|---|-----------|
| 13 Regras de contagem | 37 |
| V Variáveis aleatórias | 39 |
| 14 Distribuições de probabilidade discretas | 41 |
| 15 Distribuições de probabilidade contínuas | 43 |
| Referências | 45 |
| Apêndices | 47 |
| A Introdução ao R | 47 |
| A.1 R e RStudio IDE | 47 |
| A.2 Como baixar e instalar o R | 47 |
| A.3 Como baixar e instalar o RStudio IDE | 48 |
| A.3.1 Configurar e personalizar o RStudio IDE | 49 |
| A.4 Executar código usando o RStudio IDE | 49 |
| A.5 Objetos, funções e o operador de atribuição | 54 |
| A.5.1 Objeto moeda | 54 |
| A.5.2 Jogar uma moeda | 55 |
| A.5.3 Definindo novas funções | 57 |
| A.5.4 Argumentos | 59 |
| A.6 Projetos no RStudio | 60 |
| A.6.1 Por que usar projetos? | 60 |
| A.6.2 Como criar um projeto no RStudio | 61 |
| A.6.3 Trabalhando com projetos | 63 |
| A.7 Scripts | 63 |
| A.8 Pacotes | 64 |
| A.8.1 Instalando pacotes do CRAN | 65 |
| A.8.2 Usando um pacote | 65 |
| B Teoria ingênua dos conjuntos | 67 |
| B.1 Conjuntos | 67 |
| B.2 Operações com conjuntos | 70 |
| B.3 O conjunto vazio e o conjunto potência | 72 |

Bem-vindos

Prefácio

Parte I

Estatística e dados

Capítulo 1

Visão geral

Capítulo 2

Dados

Parte II

Visualização de dados

Capítulo 3

Tabelas

Capítulo 4

Gráficos 2D

Parte III

Medidas-resumo

Capítulo 5

Medidas de tendência central

Capítulo 6

Medidas de posição

Capítulo 7

Medidas de dispersão

Capítulo 8

Medidas de forma

Parte IV

Probabilidade

Capítulo 9

Experimento aleatório e espaço de probabilidade

Capítulo 10

Interpretações da Probabilidade

Capítulo 11

Consequências dos axiomas de probabilidade

Capítulo 12

Independência e probabilidade condicional

Capítulo 13

Regras de contagem

Parte V

Variáveis aleatórias

Capítulo 14

Distribuições de probabilidade discretas

Capítulo 15

Distribuições de probabilidade contínuas

Referências

- Chambers, John M. 2014. “Object-Oriented Programming, Functional Programming and R”. *Statistical Science* 29 (2). <https://doi.org/10.1214/13-STS452>.
- Grolemund, Garrett, e Hadley Wickham. 2015. *Hands-on programming with R*. First edition, second release. Data analysis/statistical software. Beijing Cambridge Farnham Köln Sebastopol Tokyo: O'Reilly.
- Halmos, Paul R. 2001. *Teoria ingenua dos conjuntos*. Rio de Janeiro: Editora Ciencia Moderna.
- Ismay, Chester, e Albert Young-Sun Kim. 2020. *Statistical inference via data science: a ModernDive into R and the Tidyverse*. Chapman & Hall/CRC the R Series. Boca Raton: CRC Press / Taylor & Francis Group. <https://moderndive.com/>.
- Wickham, Hadley. 2019. *Advanced R*. Second edition. Chapman & Hall/CRC: The R series. Boca Raton London New York: CRC Press, Taylor & Francis Group. <https://adv-r.hadley.nz/>.

Apêndice A

Introdução ao R

Para começar a usar o R, você precisará baixar uma cópia do R e também do RStudio IDE, um ambiente de desenvolvimento integrado (IDE) gratuito, que simplifica o uso do R. Tanto o R quanto o RStudio IDE são de código aberto, o que significa que não há custos de licenciamento envolvidos.

Antes de descrever o processo de instalação, é importante entender como o R e o RStudio IDE funcionam juntos, e também como estender sua funcionalidade usando pacotes e outros softwares complementares. (Ismay e Kim 2020, cap. 1) fornecem uma excelente explicação pedagógica desses conceitos. Esta seção se baseia na abordagem deles e utiliza as mesmas analogias.

A.1 R e RStudio IDE

O R é como o motor de um carro, enquanto o RStudio IDE funciona como o volante e o painel, conforme ilustrado na Figura A.1. Assim como um motorista interage principalmente com o volante e o painel para controlar o carro, raramente precisando interagir diretamente com o motor, o RStudio IDE também oferece uma interface amigável para trabalhar com o poderoso mecanismo do R. Essa interface simplifica o processo de usar o R para suas tarefas.

A.2 Como baixar e instalar o R

Para obter uma cópia e a versão oficial mais recente do R, acesse [The Comprehensive R Archive Network \(CRAN\)](#). O R é desenvolvido para as famílias de sistemas operacionais Unix, Windows e Mac. Na CRAN, você encontrará 3 links para baixar o R para Linux, macOS ou Windows:

- : Clique em **Download R for Linux**, escolha sua distribuição e siga as instruções de instalação específicas para sua distribuição.
- : Clique em **Download R for macOS**. Selecione o instalador que corresponda à sua versão do macOS, abra-o e siga as instruções na tela.



(a) R: Motor

(b) RStudio IDE: Painel

Figura A.1: Analogia da diferença entre e RStudio IDE

- : Clique em **Download R for Windows** e, em seguida, clique em **base**. Depois, clique no primeiro link no topo da nova página e execute o instalador. O instalador irá guiá-lo através do processo de instalação.

A.3 Como baixar e instalar o RStudio IDE

Para obter a versão oficial mais recente do RStudio IDE, acesse <https://posit.co/download/rstudio-desktop/>. Role a página para baixo, selecione seu sistema operacional e baixe o instalador apropriado. Abra o instalador e siga as instruções fornecidas.

Agora que você instalou o e o RStudio IDE, você está pronto para começar a trabalhar no RStudio IDE. Assim como você interage principalmente com o volante e o painel de um carro em vez do motor, você focará em usar a interface do RStudio IDE para trabalhar com o .

Ao abrir o RStudio IDE, você pode criar um novo script selecionando **File > New File > R Script** ou usando o atalho de teclado **Ctrl + Shift + N**. Você verá quatro painéis principais na interface, como mostrado na Figura A.2:

- **Painel 1:** É onde você escreve, edita e salva seu código R.
- **Painel 2:** Aqui você pode digitar os comandos do R e ver os resultados.
- **Painel 3:** Este painel mostra os objetos do R (como variáveis ou dados) que você criou durante a sessão atual.
- **Painel 4:** Vários elementos de saída, incluindo arquivos, gráficos e documentos de ajuda, são exibidos aqui.

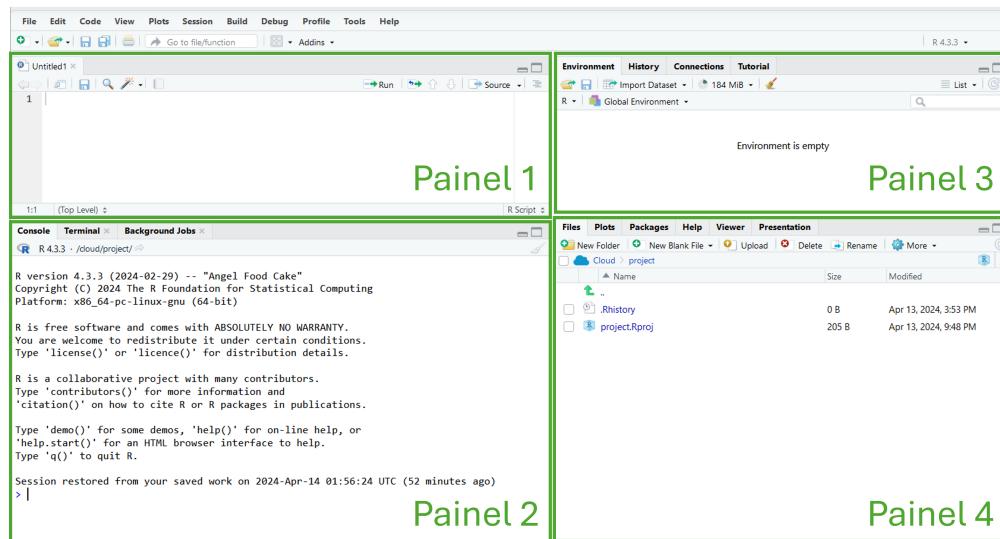


Figura A.2: Painéis do RStudio IDE

A.3.1 Configurar e personalizar o RStudio IDE

Para personalizar seu espaço de trabalho, acesse **Tools > Global Options**. Em seguida:

- Sempre inicie o **R** com uma sessão em branco, Figura A.3;
- Use o operador de pipe nativo, **|>**, Figura A.4;
- Ajuste o tamanho da fonte e selecione um tema escuro, Figura A.5;

A aplicação dessas mudanças personalizará sua experiência com o RStudio IDE, como mostrado na Figura A.6.

A.4 Executar código usando o RStudio IDE

Para nos comunicarmos com uma máquina, escrevemos instruções em uma linguagem especializada chamada código. Este código é então traduzido em um formato que a máquina entende, permitindo que ela execute as tarefas que definimos.

Vamos começar com um exemplo simples! Digite `1 + 2 + 3 + 4 + 5 + 6`, como mostrado na Figura A.7, e então aperte **Enter** or **Return** para ver o que acontece!

Se você seguiu as instruções certinho, o RStudio IDE vai mostrar o seguinte resultado:

```
1 + 2 + 3 + 4 + 5 + 6
#> [1] 21
```

Ao executar o código, você pode encontrar mensagens , avisos ou erros. Não entre em pânico! Estas são maneiras do seu computador se comunicar com você:

- **Mensagem (Message):** É simplesmente uma nota informativa. Seu código ainda será executado sem problemas.

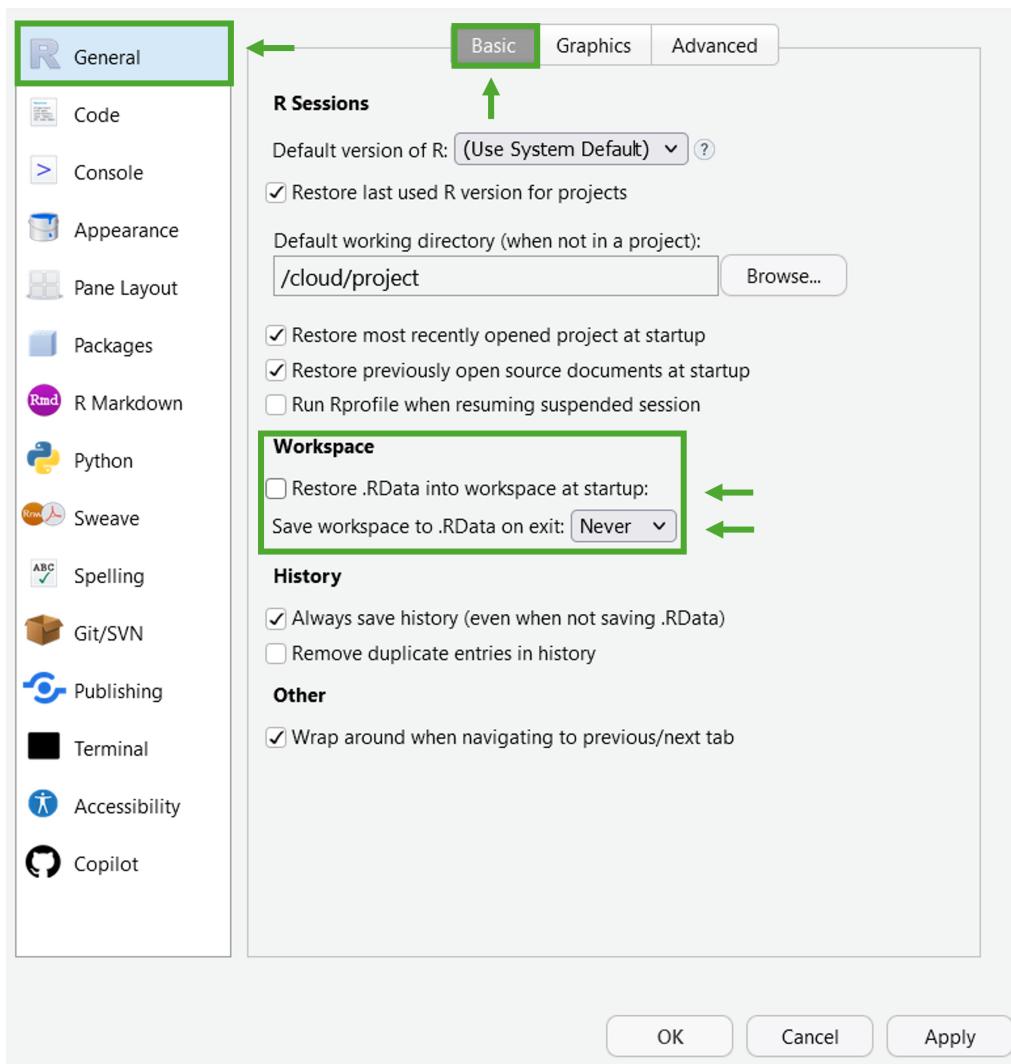


Figura A.3: Configuração do espaço de trabalho

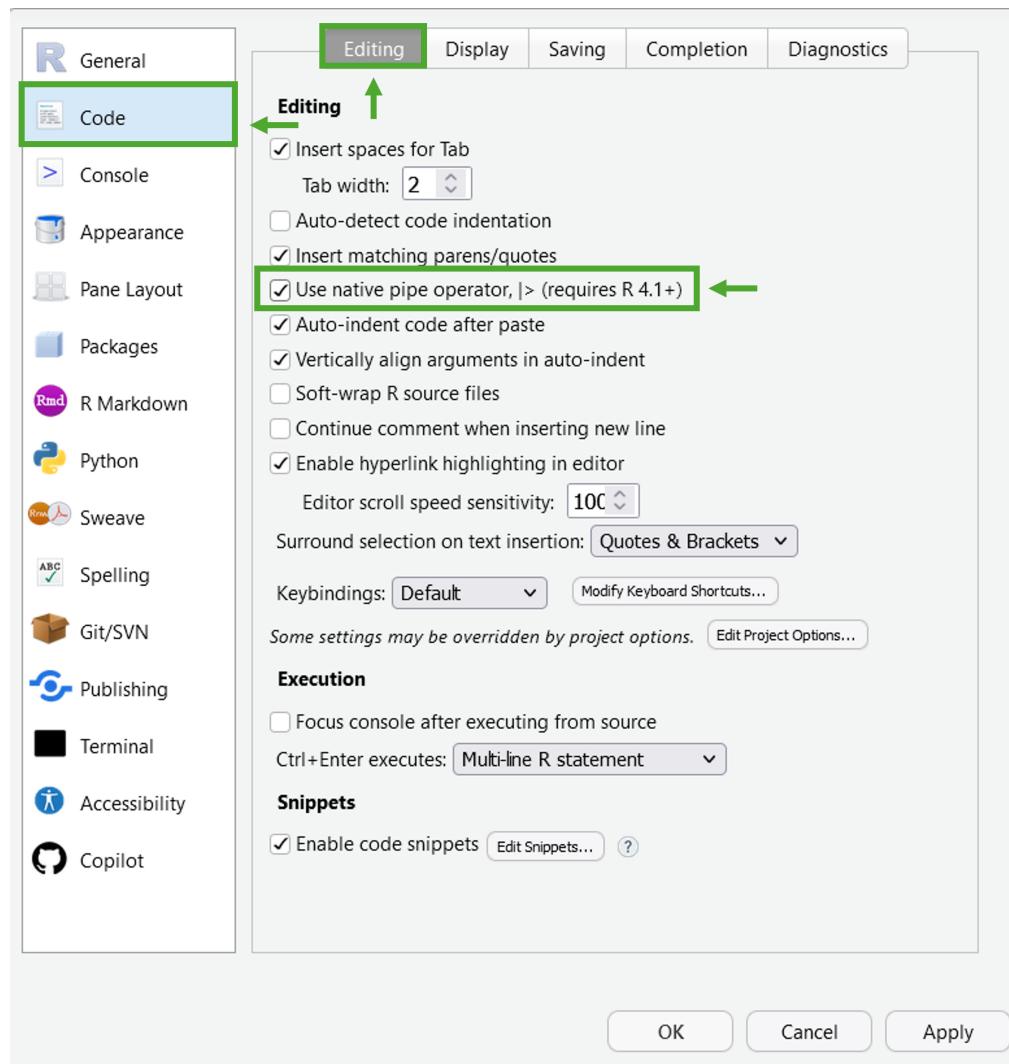


Figura A.4: Configuração para usar o operador de pipe nativo

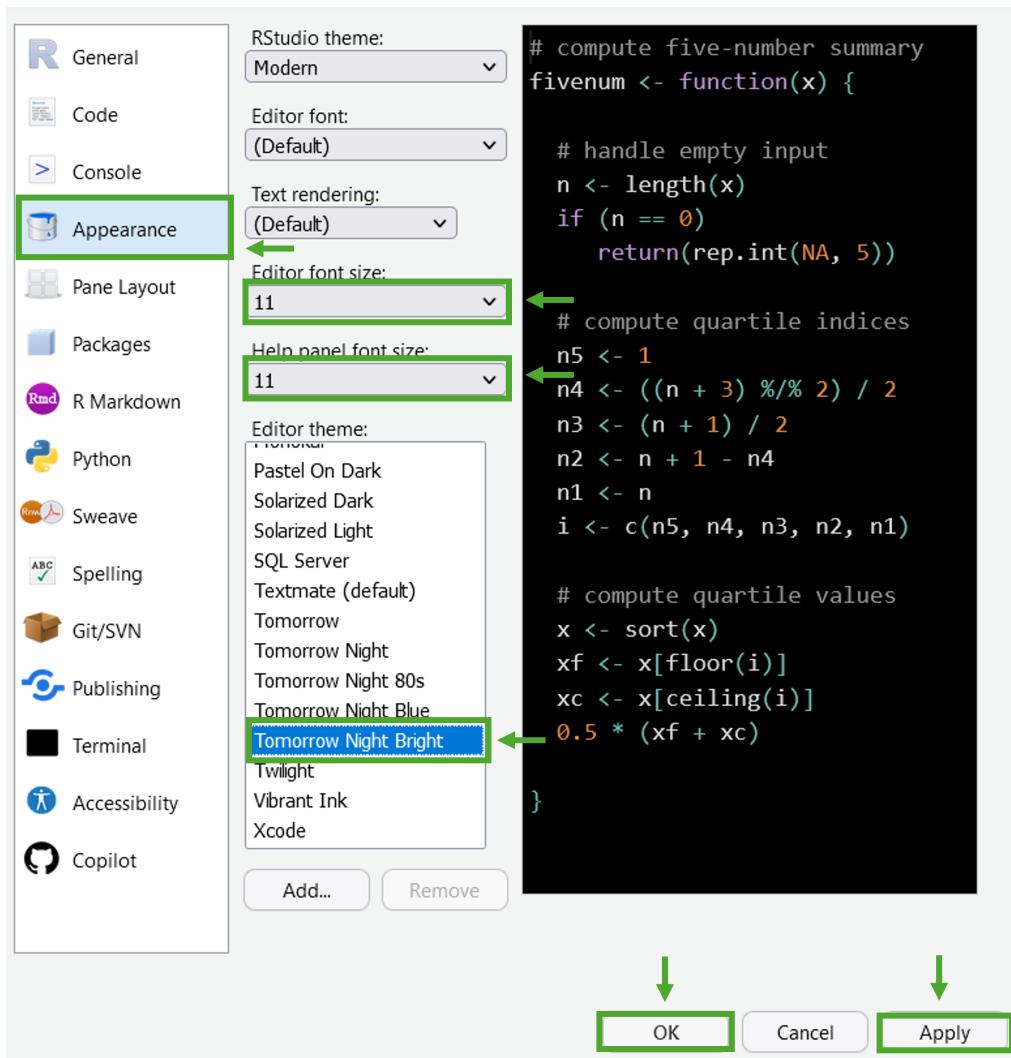


Figura A.5: Configuração para usar um tema escuro

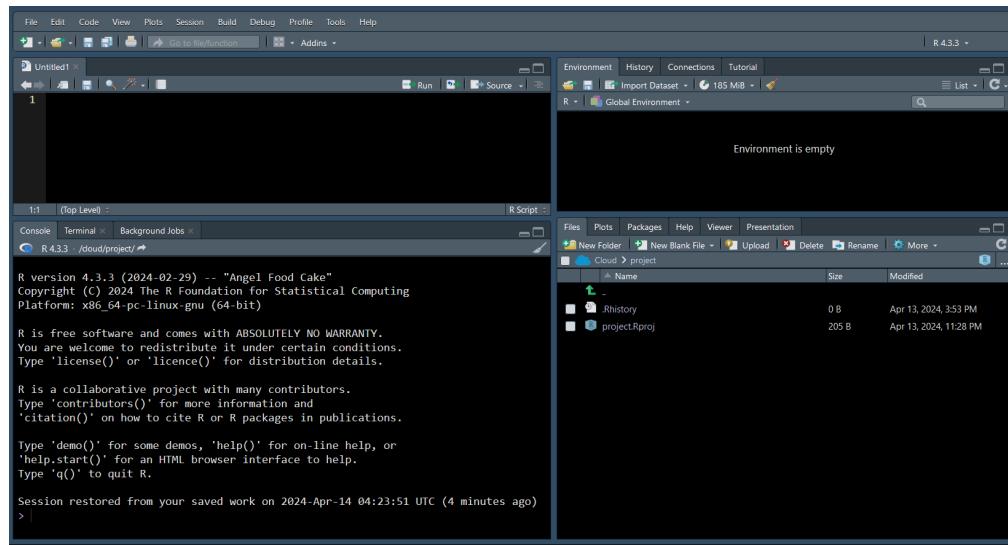


Figura A.6: Resultado da aplicação da configuração completa

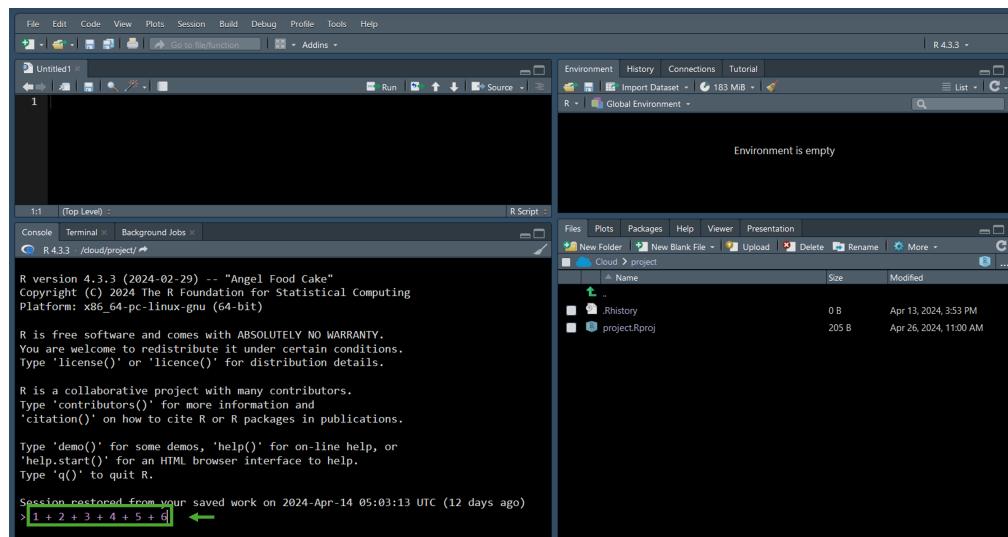


Figura A.7: Executando código no console

- **Aviso (Warning):** Alerta você sobre potenciais resultados inesperados, mas seu código ainda será executado.
- **Erro (Error):** Indica um problema fundamental que impede o seu código de ser executado. Você precisará corrigir o problema para que seu código possa funcionar.

Mais adiante, você usará o comando `install.packages()`, onde poderá encontrar mensagens. Para avisos ou erros, tente os seguintes comandos para ilustrar as diferenças:

```
log(x = -1)
#> Warning in log(x = -1): NaNs produced
#> [1] NaN

"Tudo bem" * 2
#> Error in "Tudo bem" * 2: non-numeric argument to binary operator
```

No primeiro caso, o código será executado, mas você verá o resultado `NaN` (**Not a Number**, em inglês) porque o logaritmo de um número negativo é indefinido. No segundo caso, seu código não será executado porque o  não permite a multiplicação de palavras.

Nota

No , uma string é uma coleção de um ou mais caracteres e são criadas usando aspas duplas, `""`. Por exemplo, `"Tudo bem"` é uma string.

A.5 Objetos, funções e o operador de atribuição

Tudo que existe no  é um **objeto** (Chambers 2014, 170). Isso significa que números, textos e até mesmo instruções para o computador são todos tratados como objetos. Você manipula esses objetos usando **funções**, que recebem objetos como entrada e produzem novos objetos como saída.

Por exemplo, quando você soma os números `1`, `2`, `3`, `4`, `5` e `6` (que são objetos), você usa a função `+` repetidamente para criar um novo objeto `21`. O  é construído sobre a ideia de que objetos e funções trabalham juntos.

Para trabalhar com objetos de forma simples, você pode atribuir nomes a eles usando o operador `<-`. Uma vez que um objeto tenha um nome, você pode manipulá-lo chamando esse nome.

Para entender esses conceitos de maneira prática, vamos considerar o processo de jogar uma moeda usando o .

A.5.1 Objeto moeda

Vamos imaginar uma moeda padrão com dois lados: cara e coroa, como mostrado na Figura A.8.

Para simular um lançamento de moeda em , primeiro precisamos criar um objeto que represente nossa moeda com seus dois lados: cara e coroa. Vamos atribuir um nome a esse objeto usando `<-`. Veja como combinar esses elementos em R usando a função `c()`:



Figura A.8: Os dois lados de uma moeda

```
moeda <- c("cara", "coroa")
```

Nota

Para representar os dois lados da moeda, usamos duas strings: "cara" e "coroa". Usar apenas `cara` e `coroa` sem aspas duplas poderia causar problemas, pois o as interpretaria como nomes associados a alguns objetos, em vez de valores representando os dois lados da moeda.

Agora que associamos um nome ao nosso objeto `moeda`, podemos manipulá-lo simplesmente chamando o nome `moeda`:

```
moeda
#> [1] "cara"  "coroa"
```

Quando um objeto tem um nome, ele aparecerá na aba Ambiente (**Environment**, em inglês) do painel 3 no RStudio IDE, como mostrado na Figura A.9.

A.5.2 Jogar uma moeda

Agora que você criou um objeto no , é hora de simular o lançamento de uma moeda. Uma maneira de fazer isso é com a função `sample`:

```
sample(x = moeda, size = 1)
#> [1] "coroa"
```

Aqui, o argumento `x` informa à função qual objeto usar para selecionar os elementos. O argumento `size` especifica quantos elementos escolher, neste caso `size = 1` para representar o lançamento de uma única moeda.

Vamos tentar jogar a moeda duas vezes:

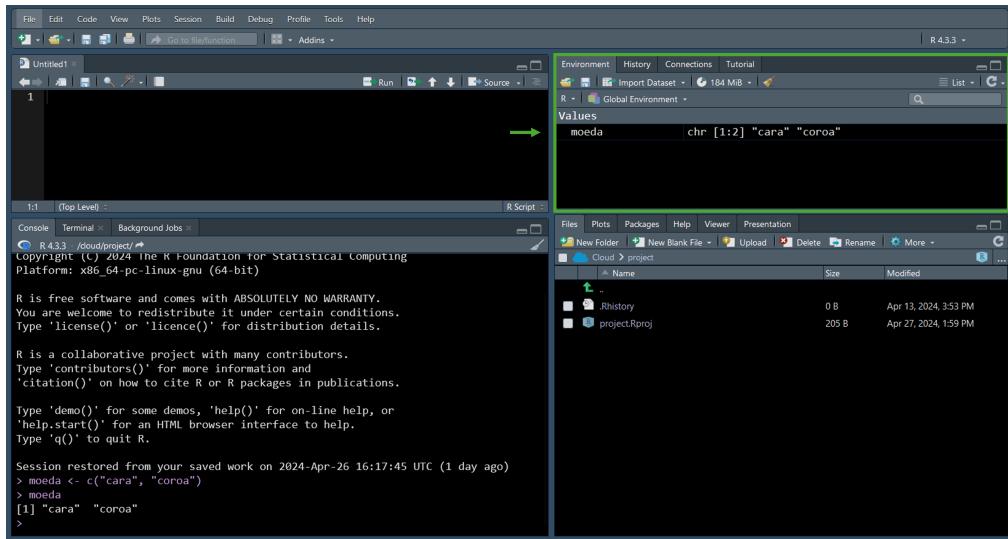


Figura A.9: Aba de ambiente

```
sample(x = moeda, size = 2)
#> [1] "cara"   "coroa"
```

E que tal 3 vezes?

```
sample(x = moeda, size = 3)
#> Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than t
```

Você encontrará um erro. Não se preocupe, os erros são úteis. A mensagem de erro menciona o argumento `replace`. Para aprender mais sobre qualquer função, use `?` seguido do nome da função, como `?sample`, conforme mostrado na Figura A.10.

Quando `replace = FALSE`, cada item em `x` só pode ser escolhido uma vez. Para simular vários lançamentos de uma moeda, em que cada resultado tem a mesma probabilidade de ser selecionado a cada vez, `replace = TRUE`:

```
sample(x = moeda, size = 3, replace = TRUE)
#> [1] "coroa" "coroa" "coroa"
```

Agora temos todos os componentes para construir uma função na próxima seção. Comecemos simulando quatro lançamentos de uma moeda:

```
lados <- c("cara", "coroa")
n_lancamentos <- 4
com_reposicao <- TRUE
lancamentos <- sample(x = lados,
                      size = n_lancamentos,
                      replace = com_reposicao)
lancamentos
#> [1] "coroa" "cara"  "cara"  "coroa"
```

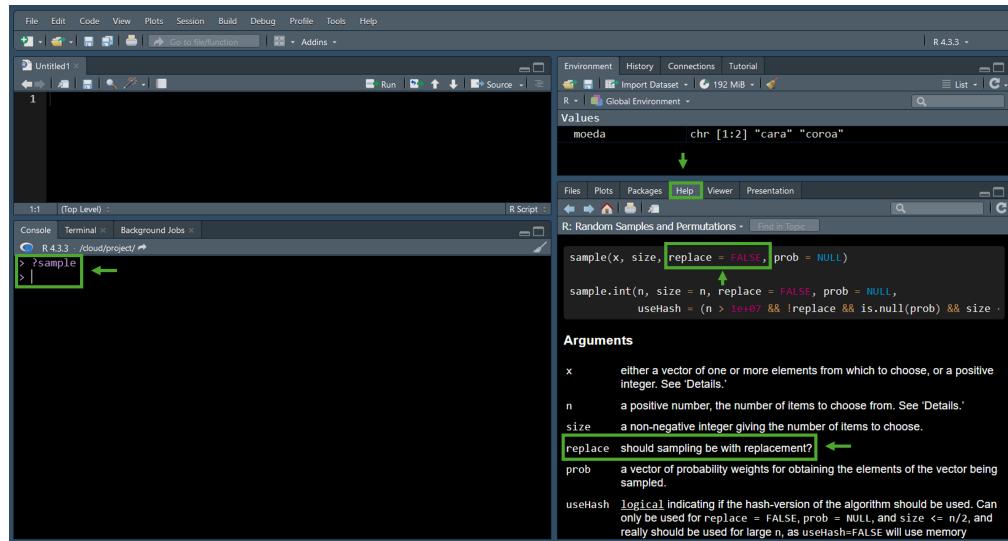


Figura A.10: Como obter ajuda no

! Importante

Para melhorar a legibilidade do código, seguiremos estas diretrizes:

- **Nomes de objetos:** Use apenas letras minúsculas sem acentos, números e sublinhados, `_`, para nomes de objetos.
- **Separação de palavras:** Utilize sublinhados para separar palavras dentro de um nome (isso é conhecido como `snake case`).
- **Nomes significativos:** Escolha nomes de objetos que refletem com precisão o que eles representam.

A.5.3 Definindo novas funções

No , uma função tem três partes básicas: um nome, um corpo e um conjunto de argumentos¹. Para criar uma, usamos a função `function`, seguindo esta estrutura:

```
nome <- function(argumentos) {
  corpo
}
```

Vamos construir nossa função de lançamento de uma moeda passo a passo. Por enquanto, começaremos sem usar argumentos:

```
jogar_moeda <- function() {

  lados <- c("cara", "coroa")
  n_lancamentos <- 4
```

¹Funções possuem complexidades adicionais, mas vamos manter a explicação simples por enquanto. Se você ficar curioso mais tarde, confira ([Wickham 2019](#), chap. 6) quando tiver mais experiência com o .

```

com_reposicao <- TRUE
lancamentos <- sample(x = lados,
                      size = n_lancamentos,
                      replace = com_reposicao)

return(lancamentos)
}

```

 Dica

Em uma função bem estruturada, usamos a declaração `return` no final do corpo para especificar explicitamente o valor que queremos que a função retorne. Isso é considerado uma boa prática na programação em .

Copie e cole o código acima no console, pressione **Enter** or **Return** e verifique a aba Ambiente (**Environment**, em inglês) do RStudio IDE, como mostrado na Figura A.11.

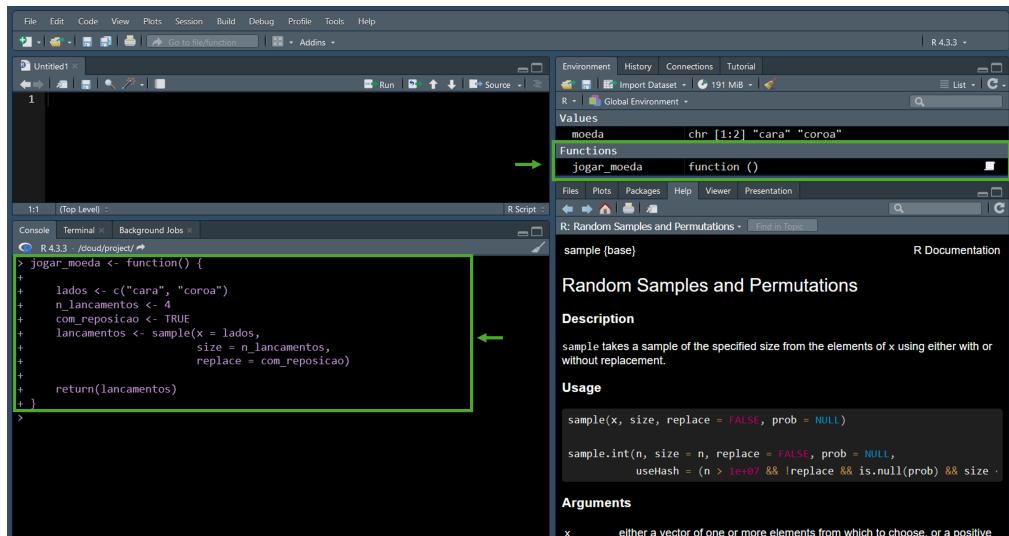


Figura A.11: Funções e aba Ambiente

Agora vamos ver se a função realmente funciona:

```
jogar_moeda()
#> [1] "coroa" "coroa" "coroa" "cara"
```

Você pode se perguntar: *Por que não usar simplesmente jogar_moeda?* Vamos tentar e ver o que acontece:

```
jogar_moeda
#> function() {
#>
#>   lados <- c("cara", "coroa")
#>   n_lancamentos <- 4
#>   com_reposicao <- TRUE
```

```
#>   lancamentos <- sample(x = lados,
#>                         size = n_lancamentos,
#>                         replace = com_reposicao)
#>
#>   return(lancamentos)
#> }
```

Nesse caso, o  vai te mostrar o código da função em vez de executá-la como faz quando você digita `jogar_moeda()`.

A.5.4 Argumentos

Para tornar nossa função mais flexível, vamos permitir que o usuário especifique o número de lançamentos de moeda. Podemos fazer isso transformando `n_lancamentos` em um argumento da função:

```
jogar_moeda <- function(n_lancamentos) {

  lados <- c("cara", "coroa")
  com_reposicao <- TRUE
  lancamentos <- sample(x = lados,
                        size = n_lancamentos,
                        replace = com_reposicao)

  return(lancamentos)
}
```

Agora, o usuário pode personalizar o número de lançamentos, especificando um valor maior ou igual a um. Vamos tentar com sete lançamentos:

```
jogar_moeda(n_lancamentos = 7)
#> [1] "coroa" "cara"  "coroa" "coroa" "cara"  "coroa" "coroa"
```

Também podemos definir valores padrão para os argumentos de uma função. Esses valores serão usados se o usuário não fornecer um valor, oferecendo flexibilidade e permitindo personalização se necessário. Vamos ilustrar essa opção definindo um valor padrão para o argumento `lados`:

```
jogar_moeda <- function(n_lancamentos,
                         lados = c("cara", "coroa")) {

  com_reposicao <- TRUE
  lancamentos <- sample(x = lados,
                        size = n_lancamentos,
                        replace = com_reposicao)

  return(lancamentos)
}
```

Se o usuário estiver satisfeito com as etiquetas “cara” e “coroa” para os lados da moeda, então podemos jogar uma moeda nove vezes da seguinte forma:

```
jogar_moeda(n_lancamentos = 9)
#> [1] "cara"  "coroa" "coroa" "cara"  "coroa" "coroa" "coroa" "coroa" "coroa"
```

No entanto, se o usuário quiser alterar o valor do argumento lados, mudando seus valores para refletir os lados da moeda mostrados na Figura A.8, podemos proceder da seguinte maneira:

```
jogar_moeda(n_lancamentos = 9, lados = c("cabeca", "navio"))
#> [1] "cabeca" "cabeca" "navio"   "cabeca" "navio"   "navio"   "navio"   "cabeca"
#> [9] "navio"
```

A Figura A.12, inspirada em (Grolemund e Wickham 2015, chap. 1), ilustra os componentes de uma função usando como exemplo `jogar_moeda`.



Figura A.12: Componentes de uma função

A.6 Projetos no RStudio

A.6.1 Por que usar projetos?

Quando você trabalha com dados (sejam reais ou simulados) ou aprende novos conceitos de estatística usando manter seus arquivos organizados é fundamental. Os Projetos do RStudio oferecem uma ótima maneira de fazer isso! Veja o que significa organizar seu trabalho em projetos:

- **Pasta de projeto dedicada:** Pense nela como um container especial para cada tópico. Todos os seus arquivos relacionados (anotações, código, dados) ficam nesta pasta.
- **Nomenclatura clara:** Dê aos seus arquivos nomes descritivos para que você possa encontrar facilmente o que precisa depois.
- **Seu ponto de partida:** Esta pasta se torna seu local de referência para qualquer coisa relacionada a esse projeto.

Os projetos do RStudio ajudam a manter seu trabalho organizado e fácil de encontrar. Além disso, você pode mover ou compartilhar facilmente um projeto inteiro, já que tudo fica junto.

Vamos começar criando um projeto chamado `estatistica_descritiva` onde organizaremos nosso trabalho sobre a função `jogar_moeda`.

A.6.2 Como criar um projeto no RStudio

Projetos no RStudio podem ser criados de três formas diferentes:

- Em um novo diretório
- Em um diretório existente.
- Clonando um repositório de controle de versão.

Vamos focar na criação de um projeto totalmente novo, então você não precisa se preocupar com as outras opções.

Para criar um novo projeto de RStudio:

- Selecione **File > New Project**
- Escolha a opção **New Directory** (Veja Figura A.13)
- Selecione **New Project**, escolha onde salvar o seu projeto e nomeie-o como `estatistica_descritiva` (Veja Figura A.14)

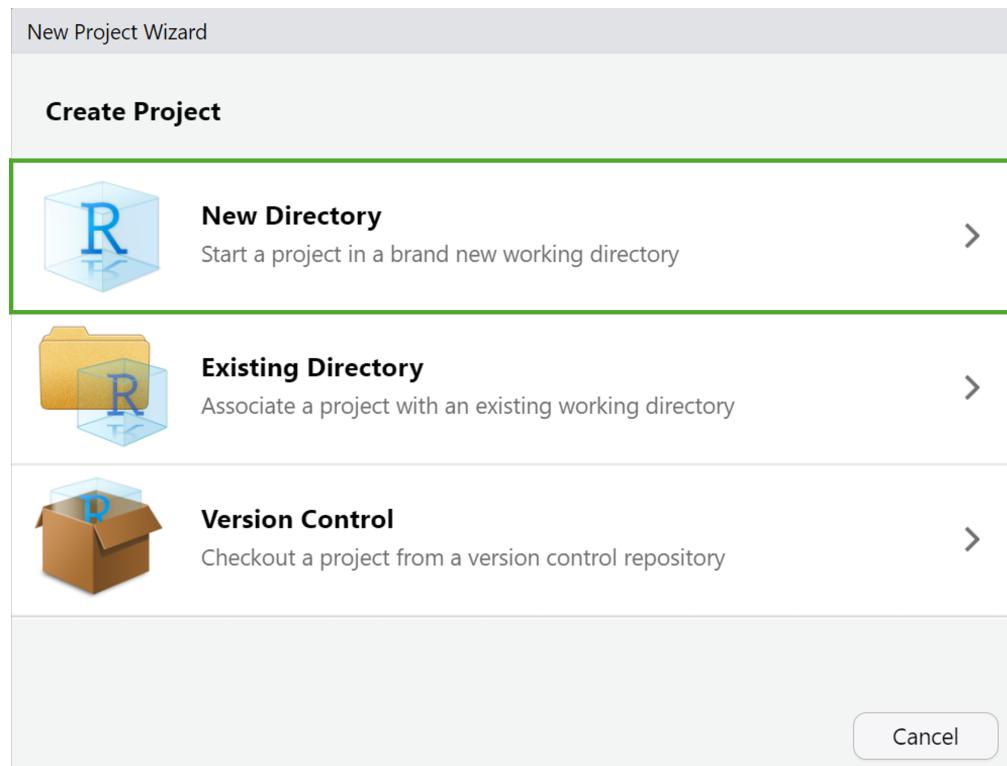


Figura A.13: Criação de um projeto

Se você seguir esses passos, uma pasta chamada `estatistica_descritiva` será criada com um arquivo `estatistica_descritiva.Rproj` dentro dela. Para manter tudo organizado, crie uma nova pasta chamada `000_scripts` dentro da pasta do projeto. A estrutura do seu projeto agora deve se parecer com a Figura A.15.

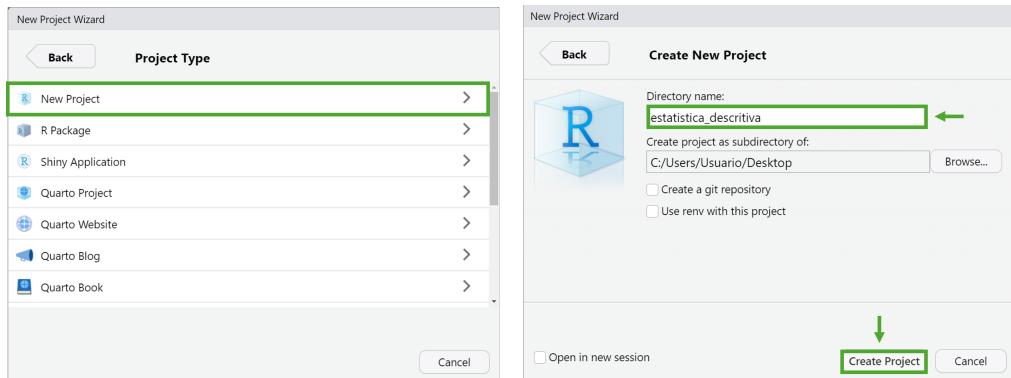


Figura A.14: Nome e localização do projeto

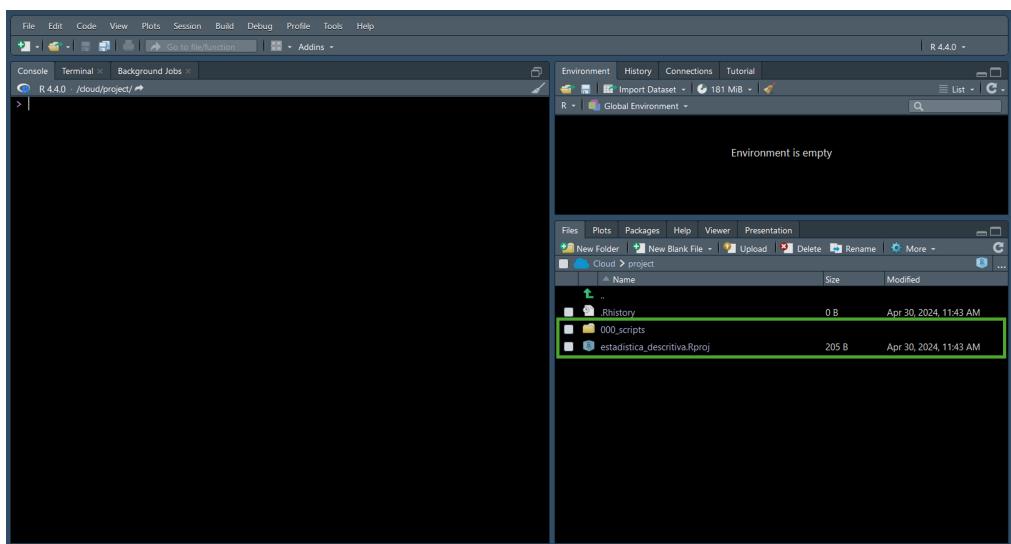


Figura A.15: Estrutura inicial do projeto

A.6.3 Trabalhando com projetos

Se fechar o RStudio IDE, para abrir o projeto e começar a trabalhar nele, é sempre necessário seguir os passos abaixo:

- Abra a pasta **estatistica_descritiva**
- Clique duas vezes no arquivo **estatistica_descritiva.Rproj**

Sempre siga estes passos para começar a trabalhar em seu projeto!

A.7 Scripts

Escrever código diretamente no console é bom para testes rápidos, mas não permite salvar seu trabalho facilmente. Scripts são essenciais porque eles:

- **Preservam o progresso:** Salve seu código para usá-lo novamente mais tarde sem ter que reescrever tudo.
- **Documentam o processo:** Adicione anotações e explicações para ajudar você (e outros!) a entender seu código no futuro.

Vamos começar criando um script para a nossa função **jogar_moeda**:

- Selecione **File > New File > R Script**.
- Copie e cole o código da função **jogar_moeda** neste script.
- Salve o arquivo na sua pasta **000_scripts** e nomeie-o como **jogar_moeda.R**.

A estrutura do seu projeto agora deve se parecer com algo assim:

```
estadistica_descritiva
 |- 000_scripts
 |   |- jogar_moeda.R
 |- estadistica_descritiva.Rproj
```

Além disso,  e o RStudio IDE tornam fácil adicionar comentários aos seus scripts das seguintes maneiras:

- **Comentários:** Comece uma linha com **#** e adicione seu comentário.
- **Seções de código:** Use quatro traços seguidos, **----**, para criar uma seção.

Vamos documentar seu script **jogar_moeda.R** apontando o propósito e o valor de saída da função **jogar_moeda**:

```
# Função jogar_moeda ----

# Propósito ----
# Simular uma série de lançamentos de uma moeda

# Saída ----
# Um vetor contendo os resultados de cada
# lançamento de uma moeda

jogar_moeda <- function(n_lancamentos,
                        lados = c("cara", "coroa")) {

  com_reposicao <- TRUE
```

```

lancamentos <- sample(x = lados,
                      size = n_lancamentos,
                      replace = com_reposicao)

return(lancamentos)
}

```

Agora, seu script está bem documentado e fácil de entender. Este processo é mostrado na Figura A.16 usando o RStudio IDE.

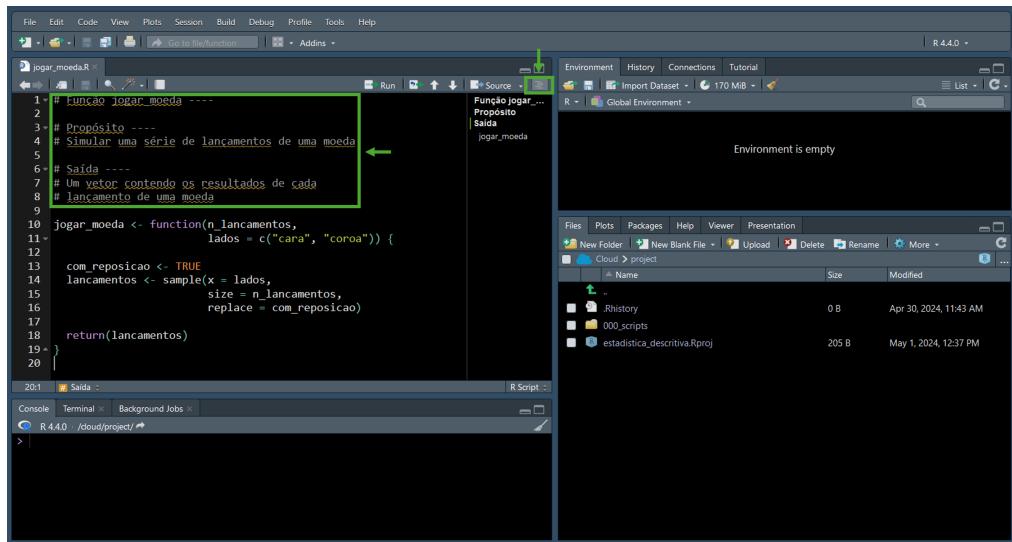


Figura A.16: Script documentado

A.8 Pacotes

Em , funções, conjuntos de dados e documentação são organizados em pacotes. Alguns deles, conhecidos como pacotes **base** vêm pré-instalados com . No entanto, existe um vasto conjunto de pacotes criados pela comunidade  que adicionam todo tipo de ferramentas adicionais.

Para usar esses pacotes criados pela comunidade, você precisará instalá-los. Esses pacotes são armazenados em repositórios, sendo o **CRAN** a principal fonte de pacotes  de propósito geral. Embora existam repositórios especializados como o **Bioconductor**, vamos nos concentrar apenas naqueles encontrados no **CRAN**.

([Ismay e Kim 2020, cap. 1](#)) têm uma analogia útil: pense em  como um smartphone novinho. Ele tem alguns aplicativos essenciais pré-instalados, mas você pode instalar uma vasta gama de novos aplicativos para realizar todos os tipos de tarefas específicas. Nesta analogia, os pacotes são como aplicativos, e os repositórios são como lojas de aplicativos.

A.8.1 Instalando pacotes do CRAN

Instalar pacotes no  é semelhante a instalar aplicativos no seu celular. Você instala uma única vez e faz atualizações ocasionais. Para instalar um pacote do **CRAN**, use a função `install.packages`, especificando o nome do pacote entre aspas duplas. Por exemplo, para instalar o pacote `tidyverse`, um conjunto popular de ferramentas para ciência de dados, você só precisa executar o seguinte código:

```
install.packages("tidyverse")
```

A.8.2 Usando um pacote

Depois de instalar o pacote `tidyverse`, vamos utilizá-lo junto com a função `jogar_moeda`. Antes de fazer isso, vamos organizar o projeto da seguinte forma:

- Crie uma pasta chamada **000_introdução_r**.
- Dentro dessa pasta, crie um script chamado **001_introdução_r_script.R**.

A estrutura do seu projeto agora deve se parecer com algo assim:

```
estadistica_descritiva
|- 000_introducao_r
|  |- 001_introducao_r_script.R
|- 000_scripts
|  |- jogar_moeda.R
|- estadistica_descritiva.Rproj
```

Pense em um pacote como um aplicativo no seu celular, você precisa abri-lo antes de usá-lo. No , usamos a função `library` para isso. Adicione este código ao seu arquivo **001_introdução_r_script.R** para usar o pacote `tidyverse`:

```
# Pacotes ----
library(tidyverse)
```

Para executar o código em seu script, clique dentro da janela do script e escolha como rodar o código:

- Linha por linha: pressione **Ctrl+Enter** (ou **Cmd+Return** no Mac).
- Tudo de uma vez: pressione **Ctrl+Shift+Enter** (ou **Cmd+Shift+Return** no Mac).

Como `jogar_moeda` não é parte de um pacote, vamos usar a função `source` para carregá-la diretamente do seu projeto:

```
# Funções ----
source(file = "000_scripts/jogar_moeda.R")
```

Execute essa linha e cheque a aba Ambiente (**Environment**, em inglês), você deve ver a função `jogar_moeda` listada. Agora, vamos usar o `tidyverse` para um pouco de diversão com visualização. Não se preocupe em entender os detalhes do código, você aprenderá mais na [?@sec-data-visualization](#). Apenas foque nos resultados:

```
# Gráfico ----
ggplot() +
  geom_bar(aes(x = jogar_moeda(n_lancamentos = 1000)))
```

Execute esse código e você deve ver um gráfico como mostrado na Figura A.17

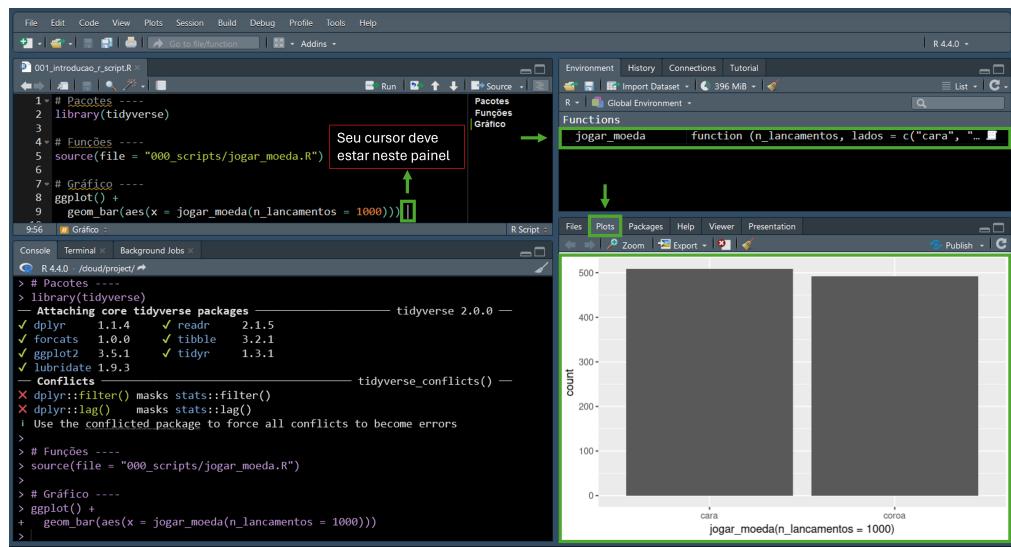


Figura A.17: Usando o pacote `tidyverse`

Apêndice B

Teoria ingênua dos conjuntos

A teoria dos conjuntos é um ramo da matemática que estuda coleções denominadas conjuntos. Compreender a teoria dos conjuntos é fundamental, pois ela serve como base para a teoria da probabilidade, que por sua vez é crucial para o estudo da estatística. No entanto, um conhecimento básico da teoria dos conjuntos é suficiente para entender os princípios fundamentais da probabilidade e da estatística, sem a necessidade de um formalismo excessivo ¹.

B.1 Conjuntos

Definição B.1 (Conjunto). Um **conjunto** é uma coleção não ordenada de elementos únicos, ou pode ser uma coleção vazia, sem nenhum elemento.

Podemos denotar um conjunto usando uma letra arbitrária como A e descrevê-lo listando seus elementos entre chaves. Por exemplo, $A = \{1, 2\}$ é o conjunto cujos elementos são os números 1 e 2. Com base em Definição B.1 e na notação anterior, é importante fazer as seguintes observações:

- $A = \{1, 2\}$ e $B = \{2, 1\}$ são o mesmo conjunto porque conjuntos são coleções não ordenadas onde a ordem não é definida.
- $C = \{1, 1, 2, 2\}$ não está bem definido porque um conjunto contém elementos únicos, onde a especificação correta seria $C = \{1, 2\}$.
- Existe um conjunto, denotado por $\emptyset = \{\}$, chamado **conjunto vazio**, que não possui elementos.
- É possível que os elementos de um conjunto sejam eles próprios conjuntos. Por exemplo, $D = \{\{1, 2\}, 3\}$ é um conjunto que contém o conjunto $\{1, 2\}$ e o número 3

O pacote **sets** do  pode ser usado para ilustrar as ideias mencionadas acima para entender o conceito de conjunto. Primeiramente, podemos criar dois conjuntos e verificar se os dois conjuntos são iguais:

¹Para uma apresentação detalhada e clara da teoria dos conjuntos usando um sistema de axiomas, você pode consultar ([Halmos 2001](#))

```
library(sets)
A <- set(1, 2)
A
#> {1, 2}
B <- set(2, 1)
B
#> {1, 2}
A == B
#> [1] TRUE
```

Também podemos verificar a propriedade de elementos únicos em um conjunto:

```
C <- set(1, 1, 2, 2)
C
#> {1, 2}
```

Além disso, podemos criar um conjunto vazio:

```
vazio <- set()
vazio
#> {}
```

Por último, podemos definir um conjunto cujos elementos podem ser conjuntos:

```
D <- set(A, 3)
D
#> {3, {1, 2}}
```

Definição B.2 (Relação de pertença). Se a é um elemento de A , expressamos essa condição como $a \in A$. Caso contrário, expressamos que a não é um elemento de A com $a \notin A$. Na teoria dos conjuntos, \in é conhecido como a relação “é um elemento de”.

Por exemplo, se $A = \{1, 2\}$ então $1 \in A$ e $3 \notin A$. Em R, podemos verificar se um elemento pertence a um conjunto usando o operador `%e%` do pacote **sets**, que representa a relação \in :

```
1 %e% A
#> [1] TRUE
3 %e% A
#> [1] FALSE
```

Às vezes, não é possível listar os elementos de um conjunto porque os elementos são infinitos ou porque não sabemos exatamente quais são. No entanto, se sabemos a propriedade que cada elemento deve ter, podemos usar uma notação matemática conhecida como **notação construtor de conjuntos** para descrever o conjunto. Essa notação é especificada como $\{x \in \Omega : P(x)\}$, onde x é um elemento genérico que pertence a Ω com a propriedade $P(x)$. Ω^2 é conhecido como **universo de discurso** e se refere ao conjunto que contém todos os elementos em consideração a partir dos quais o valor de x pode ser escolhido. Por exemplo:

- $E = \{x \in \Omega : x \text{ é um cachorro}\}$ onde E é o conjunto que contém todos os x que são cachorros. Nesse caso, $P(x)$ refere-se a ter a propriedade de ser um cachorro

² Ω é chamado Ômega

e Ω pode ser o conjunto dos seres vivos.

- $F = \{x \in \mathbb{N} : x > 5\}$ onde F contém todos os números maiores que 5 que são números naturais. Nesse caso, $P(x)$ refere-se a todos os x maiores que 5 e Ω é o conjunto dos números naturais.

Infelizmente, o **R** só pode manipular objetos que podem ser representados como números e que são finitos. Portanto, o uso de pacotes como **sets** não permite representar conjuntos como E ou F no **R**.

Definição B.3 (Subconjunto). Um conjunto A é um **subconjunto** de um conjunto B se todos os elementos de A também sejam elementos de B . Essa condição pode ser expressa através da notação $A \subseteq B$. Por outro lado, se existir um elemento que pertença a A , mas não pertença a B , representamos essa situação como $A \not\subseteq B$. Na teoria dos conjuntos, \subseteq é conhecido como a relação de “*inclusão*”.

A Figura B.1 ilustra a Definição B.3 usando um **diagrama de Venn**³.

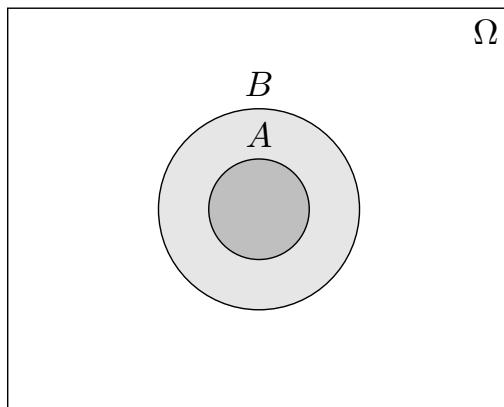


Figura B.1: $A \subseteq B$ representado por um diagrama de Venn onde Ω é o universo de discurso

Por exemplo, se $A = \{1, 2\}$ e $G = \{1, 2, 3\}$ então $A \subseteq G$ porque $1 \in A$, $1 \in G$, $2 \in A$ e $2 \in G$. No entanto, $G \not\subseteq A$ porque $3 \in G$ e $3 \notin A$. Em **R**, podemos verificar se um conjunto é um subconjunto de um conjunto usando o operador `<=` do pacote **sets**, que representa a relação \subseteq :

```
G <- set(1, 2, 3)
G
#> {1, 2, 3}
A <= G
#> [1] TRUE
G <= A
#> [1] FALSE
```

Definição B.4 (Igualdade de conjuntos). Com base em Definição B.3 podemos estabelecer uma definição equivalente para a igualdade de conjuntos. Dois conjuntos, A e B , são considerados iguais, $A = B$, se e somente se $A \subseteq B$ e $B \subseteq A$. Em outras

³Um diagrama de Venn é uma ferramenta visual que utiliza formas fechadas para representar conjuntos e ilustrar como seus elementos se relacionam.

palavras, ambos os conjuntos devem conter exatamente os mesmos elementos para serem considerados iguais.

B.2 Operações com conjuntos

Definição B.5 (União de conjuntos). A união de dois conjuntos A e B , denotada por $A \cup B$, é o conjunto de todos os elementos que estão em A ou em B . $A \cup B$ também é um conjunto e pode ser definido usando notação construtor de conjuntos como $A \cup B = \{x \in \Omega : x \in A \text{ ou } x \in B\}$

A Figura B.2 ilustra a Definição B.5 usando um diagrama de Venn.

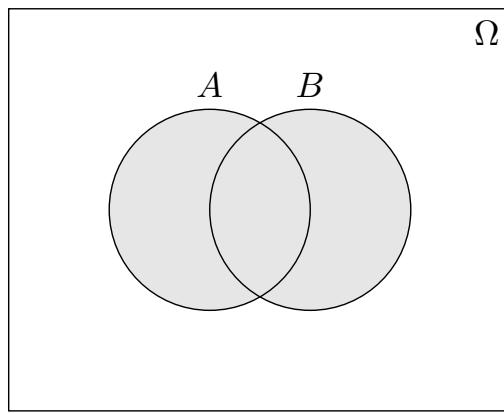


Figura B.2: $A \cup B$ representado por um diagrama de Venn onde Ω é o universo de discurso

Por exemplo, se $A = \{1, 2\}$ e $H = \{2, 3\}$, então $A \cup H = \{1, 2, 3\}$. Em R, utilizando o pacote **sets**, o operador `|` é utilizado para representar a união, \cup , de dois conjuntos da seguinte maneira:

```
H <- set(2, 3)
A | H
#> {1, 2, 3}
```

Definição B.6 (Interseção de conjuntos). A interseção de dois conjuntos A e B , denotada por $A \cap B$, é o conjunto de todos os elementos que estão em A e em B . $A \cap B$ também é um conjunto e pode ser definido usando notação construtor de conjuntos como $A \cap B = \{x \in \Omega : x \in A \text{ e } x \in B\}$.

A Figura B.3 ilustra a Definição B.6 usando um diagrama de Venn.

Por exemplo, se $A = \{1, 2\}$ e $H = \{2, 3\}$, então $A \cap H = \{2\}$. Em R, utilizando o pacote **sets**, o operador `&` é utilizado para representar a interseção, \cap , de dois conjuntos da seguinte maneira:

```
A & H
#> {2}
```

Definição B.7 (Diferença de conjuntos). A diferença de dois conjuntos A e B , denotada por $A \setminus B$, é o conjunto de todos os elementos que pertencem a A , mas não a B . $A \setminus B$

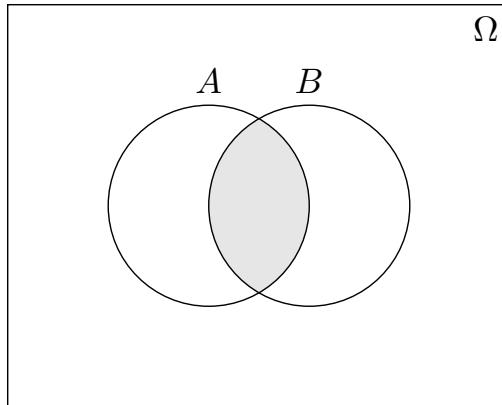


Figura B.3: $A \cap B$ representado por um diagrama de Venn onde Ω é o universo de discurso

também é um conjunto e pode ser definido usando notação construtor de conjuntos como $A \setminus B = \{x \in \Omega : x \in A \text{ e } x \notin B\}$.

A Figura B.4 ilustra a Definição B.7 usando um diagrama de Venn.

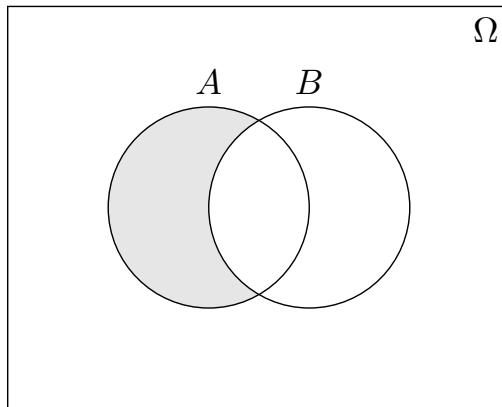


Figura B.4: $A \setminus B$ representado por um diagrama de Venn onde Ω é o universo de discurso

Por exemplo, se $A = \{1, 2\}$ e $H = \{2, 3\}$, então $A \setminus H = \{1\}$ e $H \setminus A = \{3\}$. Em R, utilizando o pacote **sets**, o operador `-` é utilizado para representar a diferença, \setminus , de dois conjuntos da seguinte maneira:

```
A - H
#> {1}
H - A
#> {3}
```

Definição B.8 (Complemento de um conjunto). Se A e Ω são conjuntos, onde Ω é o universo de discurso, o complemento de A , denotado por A^c , é o conjunto $\Omega \setminus A$. Ou seja, $A^c = \{x \in \Omega : x \notin A \text{ e } A \subseteq \Omega\}$.

A Figura B.5 ilustra a Definição B.8 usando um diagrama de Venn.

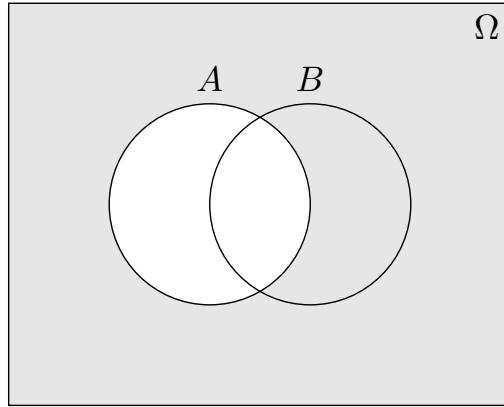


Figura B.5: A^c representado por um diagrama de Venn onde Ω é o universo de discurso

Por exemplo, se $A = \{1, 2\}$ e $\Omega = \{1, 2, 3, 4\}$, então $A^c = \{3, 4\}$. Em , utilizando o pacote **sets**, podemos determinar A^c da seguinte maneira:

```
omega <- set(1, 2, 3, 4)
omega - A
#> {3, 4}
```

B.3 O conjunto vazio e o conjunto potência

O conjunto vazio, \emptyset , tem certas características que podem parecer contraintuitivas. Em primeiro lugar, só pode haver um conjunto vazio porque quaisquer dois conjuntos que não contenham nenhum elemento são idênticos. Conforme declarado na Definição B.4, conjuntos são considerados iguais se eles têm os mesmos elementos. Como ambos os conjuntos vazios não contêm elementos, eles são considerados o mesmo conjunto.

Outra propriedade aparentemente contraintuitiva é que o conjunto vazio é um subconjunto de todos os conjuntos. Se temos um conjunto A , e afirmamos que o conjunto vazio não é um subconjunto de A , denotado por $\emptyset \subseteq A$, então, de acordo com a Definição B.3, deve existir um elemento que pertença a \emptyset , mas não a A . No entanto, como o conjunto vazio não tem nenhum elemento, é impossível que um elemento pertença a \emptyset . Portanto, a única maneira de evitar uma contradição é aceitar que o conjunto vazio é um subconjunto de todos os conjuntos, denotado por $\emptyset \subseteq A$.

Podemos resumir os resultados acima da seguinte maneira:

Teorema B.1 (Unicidade do conjunto vazio). *Só existe um conjunto vazio. Em outras palavras, se \emptyset e \emptyset' são ambos conjuntos vazios, então \emptyset é igual a \emptyset' , $\emptyset = \emptyset'$.*

Teorema B.2 (Propriedade de subconjunto do conjunto vazio). *O conjunto vazio é um subconjunto de todos os conjuntos. Para qualquer conjunto A , o conjunto vazio, \emptyset , é um subconjunto de A , $\emptyset \subseteq A$.*

Há também um conjunto chamado **conjunto das potências**. O conjunto das potências de um conjunto A , denotado como $\mathcal{P}(A)$, é um conjunto que contém todos

os subconjuntos de A . Podemos defini-lo como:

Definição B.9 (Conjunto das potências). Se A é um conjunto, então o conjunto que contém todos os subconjuntos de A , denotado como $\mathcal{P}(A)$, é definido como $\mathcal{P}(A) = \{B : B \subseteq A\}$.

Por exemplo, se $A = \{1, 2\}$ então $\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, A\} = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$ porque $\emptyset \subseteq A$ pelo Teorema B.2, $\{1\} \subseteq A$, $\{2\} \subseteq A$ e $A = \{1, 2\} \subseteq A$. Em R, podemos construir o conjunto das potências de um conjunto A , $\mathcal{P}(A)$, como 2^A utilizando o pacote **sets** da seguinte maneira:

```
potencia_A <- 2^A
potencia_A
#> {} , {1} , {2} , {1, 2}
```

