

Introdução à R e Python

Magno Severino

2024-03-03

Índice

Introdução	5
I Aprendendo R	6
1 Fundamentos de R	7
1.1 RStudio	7
1.2 Tipos de dados	8
1.2.1 O tipo de dado numérico	9
1.2.2 O tipo de dado lógico	9
1.2.3 O tipo de dado caractere	10
1.2.4 O tipo de dado fator	10
1.3 Fundamentos da linguagem	11
1.4 Variáveis	11
1.5 Verificando o tipo de uma variável	12
1.6 Estruturas de dados	14
1.6.1 Vetores	14
1.6.2 Matrizes	15
1.6.3 Listas	17
1.6.4 DataFrames	20
1.7 Exercícios	23
2 Fluxos de execução	24
2.1 Estruturas condicionais	24
2.1.1 <code>if</code> e <code>else</code>	24
2.1.2 <code>else if</code>	25
2.2 Estruturas de repetição	26
2.2.1 <code>for</code>	26
2.2.2 <code>while</code>	29
2.3 Funções	32
2.4 Pacotes	34
2.5 Exercícios	34
3 Manipulação de dados	36
3.1 Importar arquivos externos	36

3.2	O pacote tidyverse	37
3.3	O operador pipe %>%	37
3.4	Dados no formato <i>tidy</i>	38
3.5	Principais verbos do pacote dplyr	41
3.5.1	select	42
3.5.2	arrange	43
3.5.3	filter	45
3.5.4	mutate	45
3.5.5	summarise	46
3.5.6	group by	46
3.6	Funções auxiliares	48
3.7	Exercícios	52
4	Visualização de dados	55
4.1	Gramática dos Gráficos	55
4.2	O pacote ggplot	55
4.2.1	Dados	55
4.2.2	Estética	57
4.2.3	Geometria	58
4.2.4	Facetas	64
4.2.5	Coordenadas	65
4.2.6	Temas	67
4.2.7	Personalização e Estilização de Gráficos	69
4.3	Pacotes extras	73
4.3.1	O pacote patchwork	73
4.3.2	O pacote ggthemes	75
4.3.3	O pacote plotly	76
4.4	Dicas extras	77
4.5	Exercícios	77
II	Aprendendo Python	81
5	Fundamentos de Python	82
5.1	Instalação	82
5.2	Tipos de dados fundamentais	83
5.2.1	O tipo de dado inteiro	83
5.2.2	O tipo de dado ponto flutuante	84
5.2.3	O tipo de dado cadeia de caracteres	84
5.2.4	O tipo de dado lógico	85
5.2.5	Coerção de tipos	86
5.3	Objetos básicos	87
5.3.1	Listas	87

5.3.2	Tuplas	90
5.3.3	Dicionários	90
5.4	Fatias (<i>slices</i>)	92
5.5	Condicionais	93
5.6	Estruturas repetitivas	94
5.6.1	<code>for</code>	94
5.6.2	<code>while</code>	95
5.7	Comprehensions	96
5.8	Funções	97
5.8.1	Função <code>lambda</code>	98
5.9	Classes e objetos	99
5.10	Exercícios	100
6	Processamento e visualização de dados	105
6.1	Instalação de bibliotecas	105
6.2	Processamento de dados numéricos	105
6.3	Análise e processamento de dados	109
6.4	O que é pandas?	109
6.4.1	Séries	110
6.4.2	DataFrame	111
6.4.3	Principais funcionalidades	111
6.5	Dados organizados (tidy data)	117
6.6	Visualização de dados	122
6.6.1	Matplotlib	122
6.6.2	Plotnine	125
III	Estudo de casos	127
7	Estudo de casos práticos	128
	References	129

Introdução

Este livro consiste em notas de aula do minicurso de Introdução às Linguagens de Programação R e Python.

Se você está interessado nos fundamentos da linguagem R, você pode encontrar esses tópicos na primeira parte do livro. Por outro lado, se deseja explorar os fundamentos de Python, a segunda parte é onde você encontrará essas informações.

Este é um livro em constante evolução, projetado para crescer e se adaptar. Se você tiver alguma sugestão ou *feedback* sobre o conteúdo apresentado, sinta-se à vontade para enviar um e-mail para [magnetairone\[at\]gmail.com](mailto:magnetairone[at]gmail.com).

Meus sinceros agradecimentos a Luiza Tuler pela revisão do conteúdo deste livro.

Parte I

Aprendendo R

1 Fundamentos de R

1.1 RStudio

Para instalação, faça o download do R em <http://www.r-project.org>. Em seguida, instale a IDE (Integrated Development Environment) **R Studio**.

Ao abrir o RStudio, clique no menu *File/ New File/ R Script* (ou Ctrl+Shift+N). Você deve ver uma estrutura como a mostrada na figura abaixo.

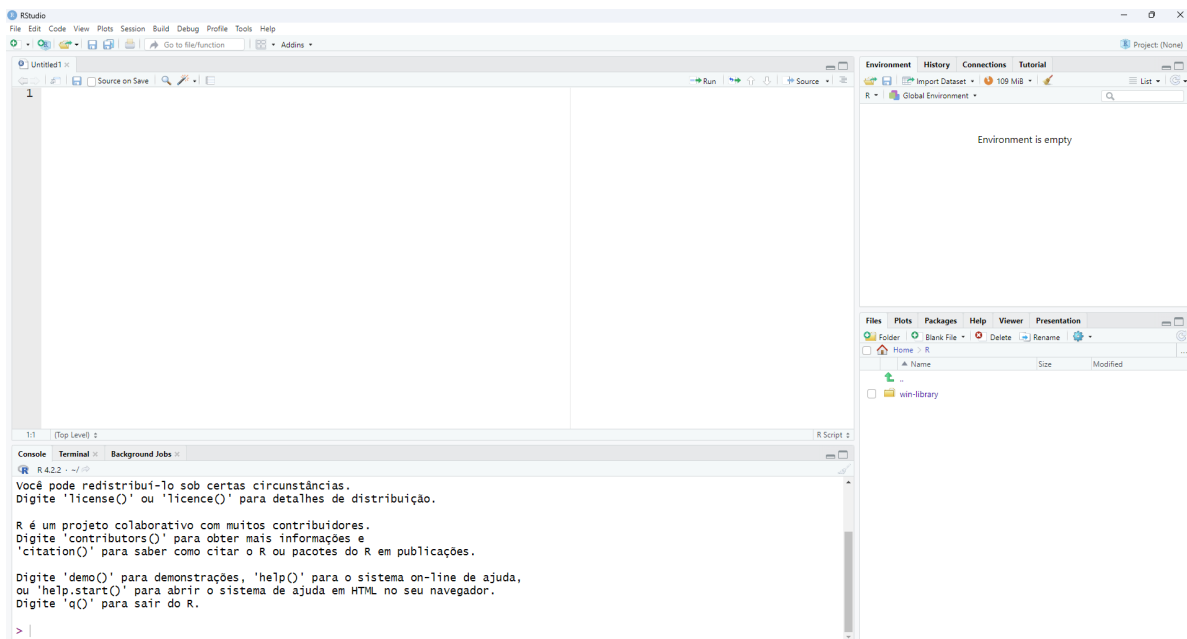


Figura 1.1: Interface do Rstudio

Note que são quatro painéis:

- *Painel de Scripts* (superior esquerdo): este painel é onde você pode escrever, editar e executar scripts R. Ele fornece recursos como destaque de sintaxe, autocompletar e verificação de código para ajudar na escrita de código.

- *Painel de Console* (inferior esquerdo): o console é onde o código R é executado e os resultados são exibidos. Você pode inserir comandos diretamente aqui e ver imediatamente os resultados. Ele também mantém um histórico de comandos executados, o que pode ser útil para referência futura.
- *Ambiente/Workspace* (superior direito): este painel exibe informações sobre os objetos (como variáveis, funções, etc.) atualmente carregados na memória do R. Ele mostra detalhes como o nome do objeto, tipo de objeto e seu valor atual. Isso é útil para monitorar e gerenciar objetos durante uma sessão de trabalho.
- *Arquivos/Plots/Pacotes/Ajuda* (inferior direito): um painel com diversas funcionalidades.
 - Arquivos: Esta guia permite navegar e gerenciar os arquivos do seu projeto. Você pode criar, renomear, excluir e organizar arquivos e pastas diretamente dentro do RStudio.
 - Gráficos (Plots): Aqui são exibidos os gráficos gerados pelo R. Quando você cria um gráfico usando funções de visualização em R, o resultado é exibido nesta guia. Isso facilita a análise visual dos seus dados e a inspeção dos gráficos durante o processo de criação.
 - Pacotes: Nesta guia, você pode visualizar e gerenciar os pacotes instalados no seu ambiente R. Ela exibe uma lista de todos os pacotes instalados, juntamente com sua versão e status (carregado ou não). Além disso, você pode instalar novos pacotes, atualizar pacotes existentes e carregar ou descarregar pacotes conforme necessário para o seu trabalho.
 - Ajuda (Help): Esta guia fornece acesso rápido à documentação e às informações de ajuda sobre funções, pacotes e outros recursos do R. Você pode pesquisar por tópicos específicos e acessar a documentação oficial diretamente no RStudio. Isso é útil para obter informações sobre a sintaxe de uma função, exemplos de uso e detalhes sobre os parâmetros disponíveis.

1.2 Tipos de dados

Sempre que estiver aprendendo uma nova linguagem, procure primeiro saber quais são os tipos de dados básicos que podem ser representados nessa linguagem.

Em R, são quatro os tipos básicos de dados disponíveis: numéricos, lógicos, caracteres e fatores.

1.2.1 O tipo de dado numérico

Os dados *numéricos* (numeric) são usados para expressar valores quantitativos, como preços, taxas e quantidades, sendo representados por números inteiros ou decimais.

```
# Número inteiro representando quantidade de acoes em uma carteira
qtd_acoes <- 100

# Número de ponto flutuante representando a taxa de inflação
taxa_inflacao <- 3.5

# Verificando a classe de taxa_inflacao
class(taxa_inflacao)
```

```
[1] "numeric"
```

A função `class()` é usada para determinar a classe de uma variável. Em outras palavras, ela fornece informações sobre o tipo de dado que uma variável representa. Nesse caso acima, a variável `taxa_inflacao` é da classe `numeric`.

1.2.2 O tipo de dado lógico

Os dados *lógicos* (logical) são empregados para representar estados ou condições, como verdadeiro ou falso, sendo úteis em operações de lógica e comparação.

```
# Verificando se a taxa de juros está aumentando
taxa_juros_aumentando <- TRUE

# Verificando se o preço das ações está caindo
queda_preco_acoes <- FALSE

# Verificando a classe de queda_preco_acoes
class(queda_preco_acoes)
```

```
[1] "logical"
```

1.2.3 O tipo de dado caractere

Já os dados do tipo *caractere* (character) são utilizados para representar texto, como nomes de países, empresas ou categorias, sendo essenciais em análises descritivas e comunicação de resultados.

```
# Nome de um país
pais <- "Brasil"

# Nome de uma empresa multinacional
empresa <- "Petróleo Brasileiro S.A."

# Verificando a classe de pais
class(pais)
```

```
[1] "character"
```

1.2.4 O tipo de dado fator

Os *fatores* (factor) são empregados para representar variáveis categóricas, como classificações, categorias ou grupos, uma forma eficiente de lidar com dados discretos e qualitativos.

```
# Classificação do risco de crédito de uma empresa
risco_credito <- factor(c("Baixo", "Médio", "Alto", "Baixo", "Alto"))

# Verificando a classe de risco_credito
class(risco_credito)
```

```
[1] "factor"
```

A função `levels()` retorna os níveis (ou categorias) de um fator. Isso é útil para entender quais são as categorias representadas pelo fator e para realizar operações de manipulação de dados com base nessas categorias.

```
# Exibindo os níveis de risco de crédito
levels(risco_credito)
```

```
[1] "Alto" "Baixo" "Médio"
```

1.3 Fundamentos da linguagem

O ambiente R refere-se ao espaço de trabalho onde todas as variáveis, funções e objetos criados durante uma sessão R são armazenados e manipulados. O ambiente inclui tanto os objetos que você criou quanto os que são carregados automaticamente por meio de pacotes ou outros mecanismos de importação de dados (mais sobre pacotes na Seção 2.4).

Por exemplo, ao usar a função `ls()` (que lista os nomes dos objetos no ambiente atual), podemos ver todos os objetos atualmente presentes no ambiente R.

```
ls()
```

Se você executou corretamente todos os comandos da Seção 1.2, deve obter como resultado no console o seguinte:

```
[1] "empresa"          "pais"              "qtd_acoes"
[4] "queda_preco_acoes" "taxa_inflacao"     "taxa_juros_aumentando"
```

usando R como calculadora numeros especiais

1.4 Variáveis

Na Seção 1.2 algumas variáveis foram criadas. Por exemplo a variável **empresa** que armazena uma cadeia de caracteres. Você viu, anteriormente a maneira de listar todas as variáveis definidas no seu ambiente. Mas, afinal, o que são variáveis?

No R, variáveis são elementos fundamentais usados para armazenar e manipular dados. Elas são como recipientes que guardam valores, objetos ou expressões. Quando você atribui um valor a uma variável, está basicamente dando um nome a esse valor para poder acessá-lo e manipulá-lo posteriormente.

Por exemplo, ao escrever `preco_acao <- 10`, você está criando uma variável chamada `preco_acao` e atribuindo a ela o valor 10. Agora, sempre que você usar `preco_acao` em seu código, estará se referindo a esse valor.

Uma prática comum escolher nomes descritivos para variáveis que ajudem a entender seu propósito ou conteúdo. Por exemplo, em um contexto econômico, você pode usar `preco_acao` para representar o preço de uma ação ou `taxa_inflacao` para representar a taxa de inflação.

Para atribuir um valor a uma variável, use o operador `<-`. O operador `=` também pode ser usado para atribuir valores a variáveis. Ambos os operadores têm o mesmo efeito prático na atribuição de valores a variáveis em R. A escolha entre eles geralmente se resume à preferência pessoal e ao estilo de codificação, embora alguns guias de estilo de código sugiram o uso do `<-`.

1.5 Verificando o tipo de uma variável

Vamos usar as funções da família `is.*` para verificar os tipos de algumas das variáveis que estão no nosso ambiente de trabalho.

- Para a variável `empresa`:

```
is.character(empresa)
```

Isso retornará `TRUE` se a variável `empresa` for do tipo caractere (`character`).

- Para a variável `pais`:

```
is.character(pais)
```

Assim como para a variável `empresa`, isso retornará `TRUE` se a variável `pais` for do tipo caractere.

- Para a variável `qtd_acoes`:

```
is.numeric(qtd_acoes)
```

Isso retornará `TRUE` se a variável `qtd_acoes` for do tipo numérico (`numeric`).

- Para a variável `queda_preco_acoes`:

```
is.logical(queda_preco_acoes)
```

Isso retornará `TRUE` se a variável `queda_preco_acoes` for do tipo lógico (`logical`).

- Para a variável `taxa_inflacao`:

```
is.numeric(taxa_inflacao)
```

Assim como para a variável `qtd_acoes`, isso retornará `TRUE` se a variável `taxa_inflacao` for do tipo numérico.

- Para a variável `taxa_juros_aumentando`:

```
is.logical(taxa_juros_aumentando)
```

Isso retornará TRUE se a variável `taxa_juros_aumentando` for do tipo lógico.

Esses exemplos ilustram como você pode usar as funções `is.*` para verificar o tipo de variáveis, ajudando a garantir que você esteja manipulando os dados corretamente em suas análises.

Outra família de funções importantes é a das funções `as.*`. Elas são usadas para fazer a coerção (converter) de um objeto de um tipo para outro. Elas permitem que você altere o tipo de dado de uma variável, o que pode ser útil em várias situações, como quando você precisa realizar operações específicas que exigem um determinado tipo de dado ou quando deseja garantir a consistência dos tipos de dados em seu código.

Algumas das funções `as.*` mais comuns incluem:

- `as.character()`: Converte um objeto para o tipo caractere (character).

```
numero <- 123
numero_caractere <- as.character(numero)
```

- `as.numeric()`: Converte um objeto para o tipo numérico (numeric).

```
texto <- "3.14"
numero <- as.numeric(texto)
```

- `as.logical()`:

```
numero <- 0
logico <- as.logical(numero)
```

Essas funções são úteis para garantir que os tipos de dados estejam corretos em seu código e para garantir que você possa realizar as operações desejadas em seus objetos. No entanto, é importante observar que nem todas as conversões podem ser bem-sucedidas, especialmente quando há perda de informações (por exemplo, ao converter de caractere para numérico). Portanto, é sempre uma boa prática verificar se a conversão foi feita corretamente e se os dados resultantes são os esperados.

Veja um exemplo de conversão de caractere para numérico com texto não numérico:

```
texto <- "abc"
numero <- as.numeric(texto)
```

Warning: NAs introduzidos por coerção

Neste exemplo, a tentativa de converter o texto “abc” para um número resultará em um valor NA (Not Available), indicando que a conversão falhou. Veja que a saída do console indica uma mensagem de *warning*.

1.6 Estruturas de dados

Em toda análise de dados, é comum lidar com conjuntos de dados que possuem diferentes estruturas e formatos. Vamos explorar quatro estruturas de dados fundamentais em R: vetor, matriz, lista e DataFrame.

1.6.1 Vetores

Um vetor em R é uma estrutura de dados unidimensional que armazena uma sequência ordenada de elementos do mesmo tipo. A função `c` nos ajuda a criar vetores.

```
# Vetor de preços de ações
precos_acoes <- c(100, 110, 105, 120, 115)
```

Em alguns casos, é de interesse definir sequências de números usando os operadores `:` e a função `seq()`.

```
# Vetor de números de 1 a 10
sequencia <- 1:10
sequencia
```

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

```
# Vetor de números de 1 a 10 com incremento de 2
sequencia_incremento <- seq(from = 1, to = 10, by = 2)
sequencia_incremento
```

```
[1] 1 3 5 7 9
```

Para verificar o tamanho de um vetor, você pode usar a função `length()`.

```
# Verificando o tamanho do vetor de preços de ações
length(precos_acoes)
```

```
[1] 5
```

```
length(1:10)
```

```
[1] 10
```

Para acessar elementos em um vetor em R, você pode usar índices numéricos ou lógicos dentro dos colchetes [].

Você pode acessar elementos usando índices numéricos dentro dos colchetes []. Por exemplo, `vetor[i]` acessa o elemento na posição `i` do `vetor`.

```
# Vetor de preços de ações
precos_acoes <- c(100, 110, 105, 120, 115)

# Acessando o segundo elemento do vetor
segundo_elemento <- precos_acoes[2]

# Acessando uma série de elementos do vetor
varios_elementos <- precos_acoes[3:5]
```

Você também pode acessar elementos usando índices lógicos dentro dos colchetes []. Por exemplo, `vetor[indices_logicos]` retorna os elementos do vetor onde os índices lógicos são `TRUE`.

```
# Acessando preços de ações maiores que 110
precos_maior_que_110 <- precos_acoes[precos_acoes > 110]
```

1.6.2 Matrizes

Uma matriz em R é uma estrutura de dados bidimensional que consiste em linhas e colunas de elementos do mesmo tipo. É útil para representar conjuntos de dados tabulares, como dados de séries temporais ou matrizes de covariância.

```
# Matriz de retornos de ativos
retornos_ativos <- matrix(c(0.05, 0.03, 0.02, 0.04, 0.06, 0.03),
                          nrow = 2, byrow = TRUE)
rownames(retornos_ativos) <- c("Ação 1", "Ação 2")
colnames(retornos_ativos) <- c("Ano 1", "Ano 2", "Ano 3")
```

O código acima cria uma matriz chamada `retornos_ativos` que armazena os retornos de dois ativos ao longo de três anos.

A função `matrix()` é usada para criar a matriz. O vetor `c(0.05, 0.03, 0.02, 0.04, 0.06, 0.03)` contém os valores dos retornos dos ativos, fornecidos em ordem de preenchimento de coluna (de cima para baixo). Os parâmetros `nrow = 2` e `byrow = TRUE` indicam que a matriz deve ter 2 linhas (para representar os dois ativos) e que os valores devem ser preenchidos por linha (ou seja, primeiro os retornos para o ano 1, depois para o ano 2 e assim por diante). As funções `rownames()` e `colnames()` são usadas para atribuir nomes às linhas e colunas da matriz, respectivamente. No caso das linhas, são atribuídos os nomes “Ação 1” e “Ação 2”, representando os dois ativos. Para as colunas, são atribuídos os nomes “Ano 1”, “Ano 2” e “Ano 3”, representando os anos em que os retornos foram registrados.

A função `class()` retorna a classe do objeto, que neste caso será “matrix”, indicando que `retornos_ativos` é uma matriz em R.

A função `dim()` retorna as dimensões da matriz, ou seja, o número de linhas e colunas.

```
# Verificando as dimensões da matriz
dim(retornos_ativos)
```

```
[1] 2 3
```

Neste caso, o resultado será `[2, 3]`, indicando que a matriz possui 2 linhas e 3 colunas.

As funções `nrow()` e `ncol()` retornam o número de linhas e colunas da matriz, respectivamente.

```
c(nrow(retornos_ativos), ncol(retornos_ativos))
```

```
[1] 2 3
```

A função `length()` retorna o número total de elementos em um objeto. Para uma matriz, isso retornará o número total de elementos, ou seja, o produto do número de linhas pelo número de colunas.

```
length(retornos_ativos)
```

```
[1] 6
```

Para acessar linhas, colunas e elementos em uma matriz em R, você pode usar índices numéricos ou nomes (se definidos). Aqui está como fazer:

- **Acessando Linhas e Colunas:** Você pode acessar linhas e colunas usando índices numéricos dentro dos colchetes `[]`. Por exemplo, `matriz[i,]` acessa a linha `i` e `matriz[, j]` acessa a coluna `j`. Para acessar uma célula específica, você usa `matriz[i, j]`, onde `i` é o número da linha e `j` é o número da coluna.

```
# Acessando a primeira linha da matriz
primeira_linha <- retornos_ativos[1, ]

# Acessando a segunda coluna da matriz
segunda_coluna <- retornos_ativos[, 2]

# Acessando o elemento na segunda linha e terceira coluna da matriz
elemento <- retornos_ativos[2, 3]

# Acessando as três primeiras linhas da matriz
primeiras_linhas <- retornos_ativos[1:2, ]

# Acessando mais de uma coluna da matriz
algumas_colunas <- retornos_ativos[, c(1,3)]
```

- **Acessando Linhas e Colunas por Nomes:** Se você definiu nomes para as linhas e/ou colunas da matriz, você pode acessá-las usando esses nomes.

```
# Acessando a linha chamada "Ação 1"
acao1 <- retornos_ativos["Ação 1", ]

# Acessando a coluna chamada "Ano 2"
ano2 <- retornos_ativos[, "Ano 2"]

# Acessando o elemento na linha "Ação 2" e coluna "Ano 3"
elemento2 <- retornos_ativos["Ação 2", "Ano 3"]
```

Em R, diferente de outras linguagens de programação, os índices de linhas e colunas em matrizes (e também em vetores, listas, etc.) começam em 1 e não em 0. Isso significa que o primeiro elemento de uma matriz está no índice 1, o segundo no índice 2, e assim por diante

1.6.3 Listas

Em R, uma lista é uma estrutura de dados flexível que pode conter elementos de diferentes tipos, como vetores, matrizes, outras listas e até mesmo funções. As listas são úteis quando você precisa armazenar e manipular conjuntos de dados heterogêneos ou estruturas complexas.

Podemos criar uma lista que armazena informações sobre um país, como seu nome, PIB, taxa de inflação e uma série temporal de valores de câmbio.

```
# Criando uma lista com informações sobre um país
pais_info <- list(
  nome = "Brasil",
  pib = 1609,
  inflacao = 0.05,
  cambio = c(4.86, 5.13, 5.20, 5.07, 4.97)
)
```

Neste exemplo, `pais_info` é uma lista que contém quatro elementos:

- `nome`: o nome do país (tipo caractere).
- `pib`: o Produto Interno Bruto do país (tipo numérico).
- `inflacao`: a taxa de inflação do país (tipo numérico).
- `cambio`: uma série temporal de valores de câmbio do país (tipo vetor numérico).

Esta lista exemplifica como podemos armazenar diferentes tipos de dados em uma lista em R. Ela pode ser usada para representar informações econômicas de um país de forma organizada e acessível.

Para acessar elementos individuais em uma lista pelo nome, usamos o operador de dólar `$`.

```
# Acessando o nome do país
pais_info$nome
```

```
[1] "Brasil"
```

```
# Acessando o PIB do país
pais_info$pib
```

```
[1] 1609
```

Também podemos acessar elementos individuais em uma lista por índice usando colchetes `[]`.

```
# Acessando o primeiro elemento da lista (nome do país)
primeiro_elemento <- pais_info[[1]]

# Acessando o terceiro elemento da lista (taxa de inflação)
```

```
terceiro_elemento <- pais_info[[3]]
```

Você deve ter notado o uso de colchetes duplos para acessar os elementos da lista. Em R, os colchetes simples (`[]`) e duplos (`[[]]`) têm diferentes propósitos quando usados para acessar elementos em uma lista.

Imagine que uma lista em R seja como um trem, e cada elemento dessa lista é um vagão do trem. Agora, dentro de cada vagão, você pode armazenar diferentes tipos de carga, como caixas, sacos ou até mesmo outros vagões.

Por exemplo, em um vagão você pode ter um vetor, em outro uma matriz e em outro apenas um número. Cada elemento da lista pode ser diferente do outro em tipo e conteúdo, assim como cada vagão de um trem pode conter coisas diferentes.

Quando você quer acessar um vagão específico do trem, você usa colchetes simples `[]`, e quando quer acessar os elementos dentro desse vagão, usa colchetes duplos `[[]]`. É como abrir a porta de um vagão para ver o que tem dentro e, em seguida, olhar dentro das caixas que estão dentro desse vagão para ver o que tem lá.

Vamos usar a analogia dos trens e vagões com a lista `pais_info`, que contém informações sobre o Brasil:

Acessando um vagão específico

Para acessar informações específicas sobre o Brasil, como o nome do país ou o valor do PIB, podemos usar os colchetes simples `[]`. Por exemplo: - `pais_info["nome"]`: obtemos o vagão que contém o nome do país, e encontramos “Brasil”. - `pais_info[2]` : obtemos o segundo vagão, neste caso é o que contém o PIB do país, e encontramos o valor 1609.

Acessando elementos dentro de um vagão

Agora, se quisermos acessar elementos específicos dentro de um vagão, usamos colchetes duplos `[[]]`. Por exemplo:

- `pais_info[["cambio"]]`: abrimos o vagão que contém informações sobre o câmbio do país e acessamos o seu conteúdo, que é um vetor com diferentes valores de câmbio ao longo do tempo.

Agora, se quisermos acessar um valor específico desse vetor, podemos usar os colchetes simples `[]` novamente:

- `pais_info[["cambio"]][3]`: isso abre o vagão que contém o vetor de câmbio e encontra o terceiro valor, que é 5.20.

Ao usar a função `class()` podemos notar a diferença entre os objetos obtidos ao usar colchetes simples e duplos.

```
class(pais_info["nome"])
```

```
[1] "list"
```

```
class(pais_info[["nome"]])
```

```
[1] "character"
```

Resumindo

Em resumo, os colchetes simples são usados para acessar subconjuntos de elementos em uma lista, preservando sua estrutura, enquanto os colchetes duplos são usados para acessar valores individuais de uma lista, sem preservar a estrutura original.

1.6.4 DataFrames

Os dataframes são estruturas de dados tabulares em R que representam conjuntos de dados retangulares onde as colunas podem ser de tipos diferentes, como numérico, caractere, lógico ou fator.

Para criar um dataframe, usamos a função `data.frame()`, fornecendo vetores de dados para cada coluna.

```
# Criando um dataframe com dados econômicos
dados_economicos <- data.frame(
  país = c("Brasil", "EUA", "China", "Índia", "Japão"),
  continente = factor(c("América", "América", "Ásia", "Ásia", "Ásia")),
  população = c(213, 328, 1441, 1380, 126),
  pib_per_capita = c(10294, 65741, 10380, 2353, 41581),
  inflação = c(0.02, 0.01, 0.04, 0.06, 0.005)
)
```

Podemos acessar elementos individuais, linhas ou colunas de um dataframe usando índices numéricos ou nomes de colunas.

```
# Acessando a primeira linha do dataframe
(primeira_linha <- dados_economicos[1, ])
```

```

      país continente população pib_per_capita inflação
1 Brasil   América      213      10294      0.02

```

```

# Acessando a coluna "país" do dataframe
(países <- dados_economicos$país)

```

```

[1] "Brasil" "EUA"    "China"  "Índia"  "Japão"

```

```

# Acessando o elemento na segunda linha e terceira coluna do dataframe
(elemento <- dados_economicos[2, 3])

```

```

[1] 328

```

Podemos combinar dataframes com base em colunas comuns usando a função `merge()`.

```

# Criando outro dataframe para junção
dados_demograficos <- data.frame(
  país = c("China", "Índia", "Japão", "Brasil", "EUA"),
  expectativa_vida = c(76, 69, 84, 75, 79)
)

# Realizando uma junção (merge) com base na coluna "país"
(dados_com_demografia <- merge(dados_economicos, dados_demograficos, by = "país"))

```

```

      país continente população pib_per_capita inflação expectativa_vida
1 Brasil   América      213      10294      0.020           75
2 China    Ásia      1441      10380      0.040           76
3 EUA     América      328      65741      0.010           79
4 Índia    Ásia      1380       2353      0.060           69
5 Japão    Ásia       126      41581      0.005           84

```

Podemos combinar novas linhas de dados ao dataframe existente.

```

# Criando outro dataframe para combinação de linhas
mais_dados <- data.frame(
  país = c("África do Sul", "Alemanha"),
  continente = c("África", "Europa"),
  população = c(60, 83),

```

```

    pib_per_capita = c(6151, 52947),
    inflação = c(0.025, NA),
    expectativa_vida = c(58, 81)
)

# Combinando os dataframes por linhas
(todos_dados <- rbind(dados_com_demografia, mais_dados))

```

	país	continente	população	pib_per_capita	inflação	expectativa_vida
1	Brasil	América	213	10294	0.020	75
2	China	Ásia	1441	10380	0.040	76
3	EUA	América	328	65741	0.010	79
4	Índia	Ásia	1380	2353	0.060	69
5	Japão	Ásia	126	41581	0.005	84
6	África do Sul	África	60	6151	0.025	58
7	Alemanha	Europa	83	52947	NA	81

! Dados faltantes

Você deve ter observado no dataframe `mais_dados` que o valor da inflação para a Alemanha está como NA. Um dado NA, abreviação de “Not Available” (não disponível), é uma marcação que indica a ausência de um valor em um conjunto de dados. No exemplo acima, a presença do dado NAn na coluna de inflação para a Alemanha significa que não há um valor disponível para a inflação desse país na tabela fornecida.

Um dado pode ser marcado como NA em diversas situações, incluindo

- **Dados ausentes:** Quando não há informação disponível para um determinado campo em um conjunto de dados. Por exemplo, falta de dados sobre o índice de desemprego em determinadas regiões devido à falta de disponibilidade ou relatórios incompletos.
- **Erros de medição ou coleta:** Em algumas situações, erros podem ocorrer durante o processo de medição ou coleta de dados, levando a valores inexatos ou ausentes. Por exemplo, ao registrar o PIB de um país, um erro humano pode levar a valores inexatos ou até mesmo ausentes em determinados trimestres devido a falhas no processo de coleta de dados.
- **Valores inaplicáveis:** Algumas variáveis podem não ser aplicáveis a todos os casos. Por exemplo, ao analisar os gastos do governo em educação, pode não haver dados disponíveis para alguns países devido a diferenças nas políticas de relatório ou à falta de investimento em educação em determinados períodos.

- **Valores não registrados:** Em algumas bases de dados, certos valores podem não ser registrados de propósito, seja por questões de privacidade ou por não serem relevantes para o contexto da análise. Por exemplo, ao coletar dados sobre o patrimônio líquido dos indivíduos em uma pesquisa de renda, alguns participantes podem optar por não divulgar suas informações financeiras por motivos de privacidade. Nesses casos, os valores correspondentes seriam marcados como NA.

1.7 Exercícios

1. Considere os seguintes setores econômicos: “Financeiro”, “Tecnologia da Informação”, “Bens Industriais” e “Saúde”. Gere aleatoriamente uma amostra de tamanho 1.000, com estes setores, com igual probabilidade de cada um ser escolhido. Mostre os primeiros valores da variável resultante e conte quantas empresas pertencem a cada setor econômico.

2 Fluxos de execução

2.1 Estruturas condicionais

O fluxo de código em R pode ser controlado por meio de estruturas condicionais, como o `if`, `else if` e `else`. Essas estruturas permitem que você execute diferentes blocos de código com base em condições específicas.

2.1.1 `if` e `else`

O `if` é uma estrutura de controle de fluxo que executa um bloco de código se uma condição especificada for verdadeira. Se a condição for falsa, o bloco de código dentro do `if` não será executado. Por outro lado, o `else` é usado para executar um bloco de código quando a condição do `if` for falsa.

A sintaxe básica do `if` e `else` em R é a seguinte:

```
if (condição) {  
  # Bloco de código a ser executado se a condição for verdadeira  
} else {  
  # Bloco de código a ser executado se a condição for falsa  
}
```

Aqui está um exemplo prático de como usar o `if` e `else` para verificar se um número inteiro escolhido aleatoriamente entre -10 e 10 é positivo ou negativo:

```
# Definindo a semente para garantir reprodutibilidade  
set.seed(42)  
  
# Gerando um número aleatório entre -10 e 10  
numero <- sample(-10:10, 1)  
  
if (numero > 0) {  
  print("O número é positivo.")  
} else {
```



```
    print("O número é negativo ou zero.")
}
```

```
[1] "O número é positivo."
```

Neste exemplo, `sample(-10:10, 1)` gera um número aleatório entre -10 e 10, e o valor é atribuído à variável `numero`. Além disso, `set.seed(123)` define a semente como 123. Isso garante que, ao gerar o número aleatório com `sample()`, o mesmo número seja escolhido sempre que o código for executado. Em seguida, verificamos se o número é positivo ou não e imprimimos a mensagem correspondente.

2.1.2 else if

Além do `if` e `else`, também podemos usar o `else if` para adicionar mais condições à estrutura condicional. O `else if` permite verificar múltiplas condições em sequência. Se a condição do `if` for falsa, ele verifica a próxima condição do `else if`. Se todas as condições do `if` e `else if` forem falsas, o bloco de código dentro do `else` é executado.

Aqui está a sintaxe do `else if`:

```
if (condição1) {
  # Bloco de código a ser executado se a condição1 for verdadeira
} else if (condição2) {
  # Bloco de código a ser executado se a condição2 for verdadeira
} else {
  # Bloco de código a ser executado se nenhuma das condições anteriores for verdadeira
}
```

Veja um exemplo prático de como usar o `if`, `else if` e `else` para avaliar o desempenho de uma empresa com base em sua receita anual:

```
# Determina a classificação da empresa com base na receita anual
receita_anual <- 1500000

if (receita_anual >= 2000000) {
  print("Empresa de Grande Porte")
} else if (receita_anual >= 1000000) {
  print("Empresa de Médio Porte")
} else if (receita_anual >= 500000) {
  print("Empresa de Pequeno Porte")
} else {
```

```
    print("Microempresa")
}
```

```
[1] "Empresa de Médio Porte"
```

Neste exemplo, a empresa é classificada com base em sua receita anual. Se a receita for igual ou superior a 2.000.000, a empresa será classificada como “Empresa de Grande Porte”. Se estiver entre 1.000.000 e 1.999.999, será classificada como “Empresa de Médio Porte”. Se estiver entre 500.000 e 999.999, será classificada como “Empresa de Pequeno Porte”. Caso contrário, será considerada uma “Microempresa”.

2.2 Estruturas de repetição

As estruturas de repetição, também conhecidas como loops, são utilizadas para executar um bloco de código repetidamente enquanto uma condição específica for verdadeira ou para percorrer uma sequência de elementos. Isso é útil quando você precisa executar uma tarefa várias vezes ou quando deseja iterar sobre uma coleção de dados.

2.2.1 for

Uma das estruturas de repetição mais comuns é o loop `for`. O loop `for` é usado para iterar sobre uma sequência de valores, como uma sequência numérica de números inteiros ou os elementos de um vetor.

Existem duas maneiras de se usar o `for` loop.

- Usando `for` para iterar sobre índices:

```
# Exemplo de loop for para iterar sobre índices
for (i in 1:5) {
  print(i)
}
```

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

Neste exemplo, o loop `for` itera sobre os valores de 1 a 5. Na primeira iteração, `i` é igual a 1; na segunda iteração, `i` é igual a 2; e assim por diante, até que `i` seja igual a 5.

- Usando `for` para iterar sobre elementos:

```
# Exemplo de loop for para iterar sobre elementos de um vetor
clientes <- c("João", "Maria", "José", "Ana")
for (nome in clientes) {
  print(nome)
}
```

```
[1] "João"
[1] "Maria"
[1] "José"
[1] "Ana"
```

Neste exemplo acima, o loop `for` itera sobre os elementos do vetor `clientes`. Na primeira iteração, `nome` é igual a “João”; na segunda iteração, `nome` é igual a “Maria”; e assim por diante, até que todos os elementos do vetor sejam percorridos.

Em ambos os exemplos, o bloco de código dentro do loop `for` é executado repetidamente para cada valor de `i` (no primeiro exemplo) ou `nome` (no segundo exemplo) até que a sequência seja completamente percorrida.

No exemplo abaixo, vamos simular dados econômicos para 10 países fictícios e calcular o PIB per capita de cada país.

```
set.seed(42)
pib_paises <- runif(10, min = 25000000, max = 40000000)
populacao_paises <- runif(10, min = 1000000, max = 15000000)

pib_per_capita <- numeric(length = 10)

# Loop for para calcular o PIB per capita para cada pais
for (i in 1:10) {
  # Calculando o PIB per capita
  pib_per_capita[i] <- pib_paises[i] / populacao_paises[i]
}
print(round(pib_per_capita, 3))
```

```
[1] 5.227 3.529 2.080 8.185 4.634 2.315 2.453 10.216 4.556 4.022
```

💡 Clique e veja um exemplo extra

Imagine que temos uma série temporal representando o preço de fechamento diário de uma ação ao longo de um período de 30 dias. Queremos calcular a média móvel de 5 dias desse preço, ou seja, para cada dia, queremos calcular a média dos preços de fechamento dos cinco dias anteriores, incluindo o dia atual.

Primeiro, vamos simular os dados do preço de fechamento diário da ação:

```
set.seed(42)
preco_acao <- runif(30, min = 9, max = 15)
```

Agora, vamos calcular a média móvel de 5 dias usando um loop for:

```
media_movel <- numeric(length = 26) # Vetor para armazenar a média móvel

for (i in 5:30) {
  media_movel[i - 4] <- mean(preco_acao[(i - 4):i])
}
```

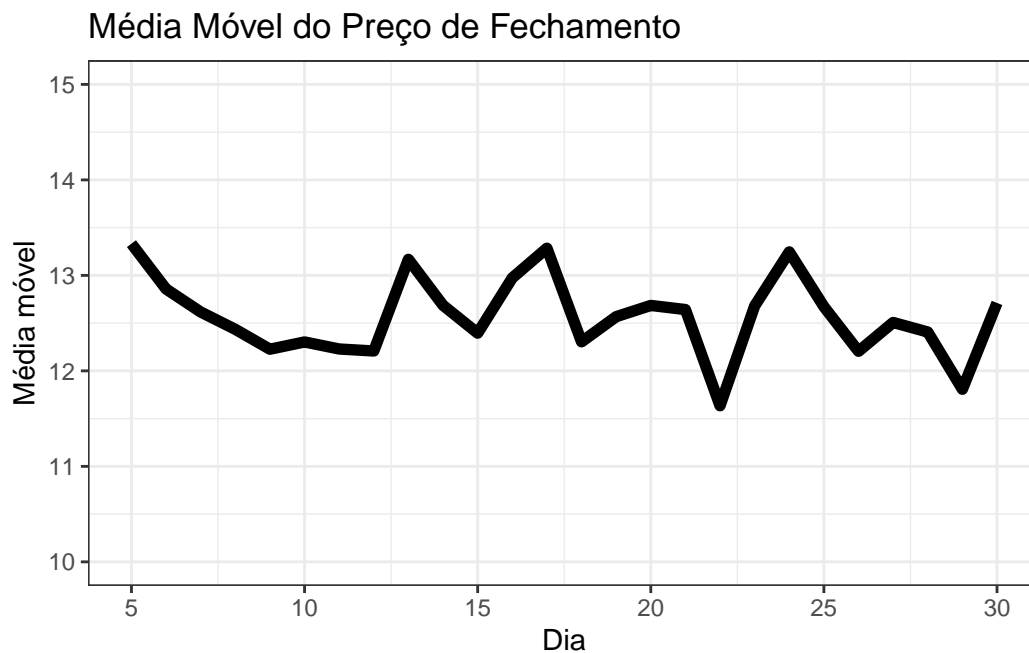
Neste loop for, começamos a partir do quinto dia, pois precisamos de pelo menos cinco dias para calcular a média móvel de 5 dias. Para cada dia a partir do quinto dia até o trigésimo dia, calculamos a média dos preços de fechamento dos cinco dias anteriores, incluindo o dia atual, e armazenamos esse valor no vetor `media_movel`.

Agora, podemos imprimir a média móvel calculada:

```
print(media_movel)
```

```
[1] 13.33226 12.85740 12.61682 12.43505 12.22691 12.30289 12.22926 12.20829
[9] 13.16830 12.68642 12.39510 12.97382 13.28476 12.30414 12.56762 12.68527
[17] 12.64209 11.63467 12.68036 13.24636 12.67289 12.20510 12.50690 12.40711
[25] 11.80747 12.71175
```

Este exemplo demonstra como usar um loop for em conjunto com vetores para calcular a média móvel de uma série temporal. O gráfico abaixo mostra a média móvel ao longo dos dias.



i Nota

Você vai aprender a construir gráficos como este no [Capítulo 4](#).

2.2.2 while

A estrutura **while** é usada para repetir um bloco de código enquanto uma condição especificada for verdadeira. Aqui está a estrutura geral de um loop while:

```
while (condição) {  
    # Código a ser repetido enquanto a condição for verdadeira  
}
```

A condição é uma expressão lógica que é avaliada antes de cada execução do bloco de código dentro do loop. Se a condição for verdadeira, o bloco de código é executado; se a condição for falsa, o loop é interrompido e o controle é passado para a próxima linha de código após o loop.

No exemplo abaixo, vamos definir um vetor chamado **acoes**, que contém uma lista de atividades possíveis que uma pessoa pode realizar durante o dia. Dentre as ações possíveis, uma será escolhida aleatoriamente.

```

acoes <- c( "Aprender a programar em R",
             "Aprender a programar em Python",
             "Fazer um café",
             "Descansar")

set.seed(42)
acao <- sample(acoes, 1)
print(acao)

```

```
[1] "Aprender a programar em R"
```

No trecho de código a seguir, usamos a estrutura **while** para continuar selecionando aleatoriamente uma atividade do vetor **acoes** até que a atividade selecionada seja “Descansar”. O loop começa verificando se a variável **acao** é diferente de “Descansar”. Se essa condição for verdadeira, uma nova atividade é selecionada aleatoriamente do vetor **acoes** usando a função **sample()** com **size = 1**, o que significa que estamos selecionando apenas um elemento aleatório do vetor. Em seguida, a atividade selecionada é impressa na tela usando a função **print()**. Esse processo se repete até que a atividade selecionada seja “Descansar”, momento em que o loop é encerrado.

```


set.seed(420)
while(acao != "Descansar") {
  acao <- sample(acoes, 1)
  print(acao)
}

```

```

[1] "Aprender a programar em R"
[1] "Aprender a programar em R"
[1] "Aprender a programar em Python"
[1] "Aprender a programar em Python"
[1] "Descansar"

```

 Clique e veja um exemplo extra

Vamos considerar um exemplo onde queremos simular a evolução de uma população ao longo do tempo, onde não sabemos exatamente quantos períodos serão necessários para que a população atinja um determinado limite. Neste caso, usaremos um loop **while** para continuar simulando o crescimento populacional até que a população alcance um certo valor limite.

```

set.seed(42) # Define uma semente para a replicabilidade dos resultados

# População inicial
populacao <- 1000

# Taxa de crescimento anual da população (em decimal)
taxa_crescimento <- 0.02

# População limite desejada
limite_populacional <- 2000

# Inicializando o contador de anos
anos <- 0

# Simulando o crescimento populacional até atingir o limite
while (populacao < limite_populacional) {
  # Calculando o número de novos indivíduos neste ano
  novos_individuos <- populacao * taxa_crescimento

  # Incrementando a população com os novos indivíduos
  populacao <- populacao + novos_individuos

  # Incrementando o contador de anos
  anos <- anos + 1
}

# Imprimindo o número de anos necessários para atingir o limite populacional
print(paste("Foram necessários", anos, "anos para atingir uma população de", populacao))

```

```
[1] "Foram necessários 36 anos para atingir uma população de 2039.8873437157"
```

i Nota

É possível calcular diretamente o número de anos necessários para atingir a população limite desejada. Com um pouco de álgebra você chege na seguinte fórmula para calcular o número de anos:

$$\text{anos} = \frac{\log\left(\frac{\text{limite_populacional}}{\text{populacao_inicial}}\right)}{\log(1 + \text{taxa_crescimento})}.$$

2.3 Funções

Uma função em R é um bloco de código que realiza uma tarefa específica e pode ser reutilizado várias vezes. A sintaxe para definir uma função em R segue o padrão:

```
nome_da_funcao <- function(parametros) {  
  # Corpo da função  
  # Código que realiza a tarefa desejada  
  # Pode incluir operações matemáticas, manipulação de dados, etc.  
  return(resultado) # Retorna o resultado desejado  
}
```

Os parâmetros são variáveis que uma função recebe como entrada para executar suas operações. Eles são especificados entre parênteses na definição da função. Dentro do corpo da função, os parâmetros podem ser utilizados para realizar cálculos ou operações.

O exemplo abaixo define uma função para realizar uma regressão linear simples. A função `regressao_linear` recebe dois parâmetros: `x` e `y`, que representam os dados de entrada para a regressão linear. Dentro da função, um modelo de regressão linear é criado usando a função `lm()` do R com os dados `y` em função de `x`. A documentação da função `lm()` pode ser acessada ao executar o comando `?lm`. O modelo resultante é retornado como resultado da função.

```
# Função para realizar regressão linear simples  
regressao_linear <- function(x, y) {  
  modelo <- lm(y ~ x) # Criando o modelo de regressão linear  
  return(modelo) # Retornando o modelo  
}  
  
# Dados de exemplo: salário (y) em função dos anos de educação (x)  
anos_educacao <- c(10, 12, 14, 16, 18)  
salario <- c(2500, 3300, 3550, 3700, 4500)  
  
# Chamando a função de regressão linear  
modelo_regressao <- regressao_linear(anos_educacao, salario)
```

Veja o sumário com o resultado do modelo treinado.

```
# Exibindo os resultados da regressão  
summary(modelo_regressao)
```

Call:


```
lm(formula = y ~ x)
```

Residuals:

1	2	3	4	5
-130	230	40	-250	110

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	430.00	498.20	0.863	0.45156
x	220.00	34.88	6.307	0.00805 **

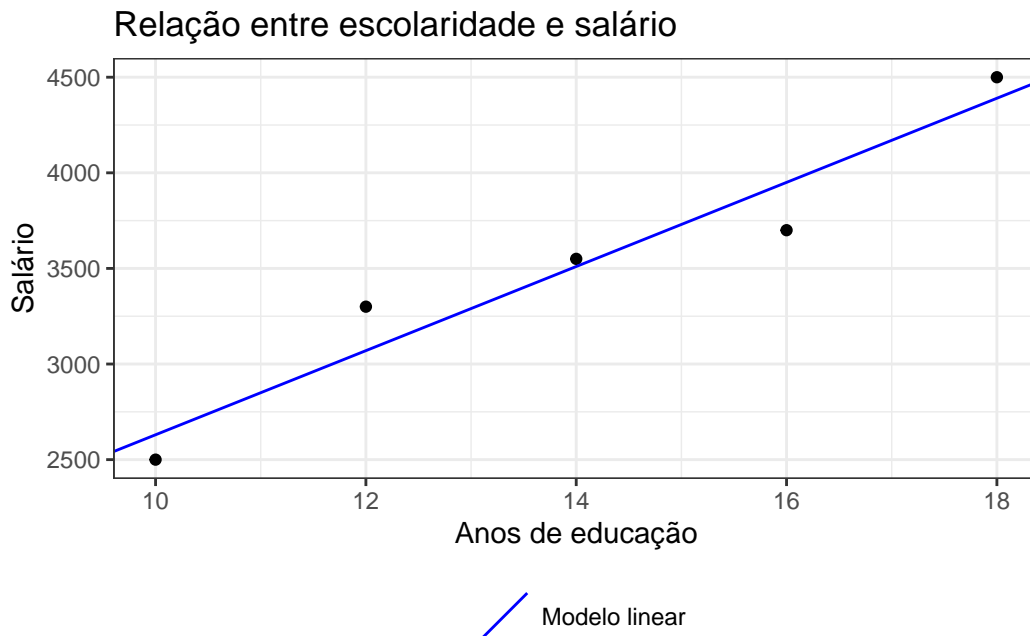
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 220.6 on 3 degrees of freedom

Multiple R-squared: 0.9299, Adjusted R-squared: 0.9065

F-statistic: 39.78 on 1 and 3 DF, p-value: 0.008054

A figura abaixo mostra um gráfico de dispersão que representa a relação entre anos de educação e salário. A reta azul mostra o modelo de regressão linear treinado com os dados.



Nota

Você vai aprender a construir gráficos como este no [Capítulo 4](#).

2.4 Pacotes

Os pacotes no R são conjuntos de funções, conjuntos de dados e documentação que ampliam as capacidades básicas da linguagem R. Eles são essenciais para expandir a funcionalidade do R, permitindo que os usuários realizem uma ampla variedade de tarefas.

Os pacotes do R estão disponíveis no CRAN (Comprehensive R Archive Network), um repositório centralizado que abriga uma vasta coleção de pacotes, manuais, documentações e outros recursos relacionados ao R. O CRAN é essencialmente o principal hub para distribuição de pacotes R e serve como uma fonte confiável e abrangente de ferramentas para análise de dados, estatísticas e muito mais. Para acessar os pacotes do CRAN, você pode usar a função `install.packages()`. Por exemplo:

```
install.packages("nome_do_pacote")
```

Depois de instalar um pacote, você precisa carregá-lo em sua sessão R usando a função `library()`:

```
library(nome_do_pacote)
```

Isso tornará as funções e conjuntos de dados do pacote disponíveis para uso em sua sessão atual.

2.5 Exercícios

1. Neste exercício, vamos simular o lançamento de uma moeda e armazenar os resultados como um fator contendo os níveis “cara” e “coroa”. Para isso, siga os passos abaixo:

a) Utilize o comando abaixo para gerar amostras aleatórias seguindo a distribuição binomial para simular o lançamento de 100 moedas:

```
set.seed(42)
lancamentos <- rbinom(100, 1, 0.5)
```

b) Considere que 0 represente “cara” e 1 represente “coroa”. Crie uma variável para armazenar os lançamentos como um fator contendo os níveis “cara” e “coroa”.

c) Conte quantas vezes cada um dos resultados ocorreu neste experimento.

d) Utilize um loop para percorrer o vetor de trás para frente e descubra qual foi o último lançamento que resultou em uma “cara”.

2. Trabalhando com dados de pacotes.

a) Instale o pacote `nycflights13` utilizando o comando abaixo:

```
install.packages("nycflights13")
```

b) Carregue no seu ambiente o pacote instalado:

```
library(nycflights13)
```

c) Utilizando os comandos abaixo, verifique o conteúdo dos dataframes `flights` e `airports`:

```
?flights  
?airports
```

d) Filtre os voos que aconteceram em 25/01/2013 e armazene-os na variável `natal`;

e) Quantos voos partiram de Nova Iorque em 25/12/2013?

f) Obtenha um sumário da coluna `dep_delay`. Há dados faltantes? Se sim, remova-os.

g) Obtenha o nome do aeroporto de destino do voo com maior atraso de partida em 25/12/2013. Dica: mescle os dados de `flights` e `airports`.

3 Manipulação de dados

Dominar técnicas de manipulação e processamento de dados em R é fundamental para qualquer pessoa que trabalhe com análise de dados ou ciência de dados. A capacidade de limpar, transformar e preparar dados de maneira eficiente é essencial para garantir que os resultados das análises sejam precisos e confiáveis. Além disso, o domínio dessas técnicas economiza tempo e aumenta a produtividade.

3.1 Importar arquivos externos

Dois dos formatos mais comuns para armazenamento de dados não tão grandes são `csv` e `xlsx`.

Ao carregar dados de arquivos `csv` em R, duas opções comumente utilizadas são as funções `read.csv()` e `read_csv()`. Ambas são eficazes para importar dados tabulares, mas apresentam diferenças significativas. A função `read.csv()` é uma opção padrão no R base, sendo simples de usar e amplamente conhecida. Por outro lado, `read_csv()` faz parte do pacote `readr`, oferecendo desempenho otimizado e detecção automática de tipos de dados. Enquanto `read.csv()` tende a ser mais lenta, especialmente com grandes conjuntos de dados, `read_csv()` é mais rápida e precisa, sendo capaz de manter os nomes das colunas como símbolos e converter adequadamente os dados, inclusive lidando com strings vazias.

```
# Usando read.csv()
dados_read_csv <- read.csv("dados.csv")

# Usando read_csv() do pacote readr
library(readr)
dados_readr <- read_csv("dados.csv")
```

Para importar dados de um arquivo Excel (formato `xlsx`) em R, podemos usar a biblioteca `readxl`. Primeiro, é necessário instalá-la usando o comando `install.packages("readxl")`. Em seguida, podemos usar a função `read_excel()` para ler os dados. Por exemplo:

```
library(readxl)
dados <- read_excel("arquivo.xlsx")
```

Definindo o seu diretório de trabalho

É uma boa prática definir um diretório de trabalho em seus scripts R porque isso ajuda a manter a organização e facilita o acesso aos arquivos de dados e resultados. Ao definir um diretório de trabalho, você garante que todos os arquivos referenciados em seus scripts serão encontrados facilmente, sem a necessidade de especificar caminhos absolutos longos. Para definir o diretório de trabalho no R, você pode usar a função `setwd()`. Por exemplo, se você deseja definir o diretório como “C:/MeuDiretorio”, você pode fazer o seguinte:

```
setwd("C:/MeuDiretorio")
```

Você pode definir o diretório usando a interface do RStudio. Basta selecionar no menu “Session” a opção “Set Working Directory” e em seguida “Choose Directory”. Isso abrirá uma caixa de diálogo onde você pode navegar até o diretório desejado e selecioná-lo. Depois de selecionar o diretório, ele se tornará o diretório de trabalho atual.

3.2 O pacote tidyverse

O pacote `tidyverse` é uma coleção de pacotes do R projetados para trabalhar de forma integrada e intuitiva na análise de dados. Ele inclui uma variedade de pacotes poderosos e populares, como `ggplot2`, `dplyr`, `tidyr`, `tibble`, `readr`, `purrr`, `forcats` e `stringr`. Cada pacote no `tidyverse` foi projetado para lidar com uma etapa específica do fluxo de trabalho de análise de dados, desde a importação e limpeza até a visualização e modelagem. Todos os pacotes no `tidyverse` compartilham uma filosofia de design subjacente, gramática e estruturas de dados, veja mais [na página do pacote](#).

3.3 O operador pipe %>%

O operador `%>%`, conhecido como pipe, é uma ferramenta poderosa em R que facilita a encadeamento de operações em sequência. Ele permite escrever código de forma mais clara e concisa, especialmente ao trabalhar com pacotes do `tidyverse`. O pipe recebe o resultado de uma expressão à esquerda e o passa como primeiro argumento para a próxima expressão à direita.

Dica

Você não precisa digitar `%>%` toda vez que precisar. Utilize o atalho `Ctrl+Shift+M`.

Vamos supor que temos uma função `f`, uma função `g` e uma variável `x`. Queremos aplicar `g` a `x` e, em seguida, aplicar `f` ao resultado. Aqui está como poderíamos fazer isso de duas maneiras: usando a abordagem encadeada tradicional e usando o pipe `%>%`.

```
resultado <- f(g(x))

x %>%
  g() %>%
  f()
```

Ambos os métodos produzirão o mesmo resultado. No entanto, a segunda abordagem usando o pipe `%>%` é mais legível e fácil de entender, especialmente quando estamos encadeando múltiplas operações. Isso torna o código mais conciso e mais próximo de uma leitura natural da operação que está sendo realizada.

Dica

Uma boa prática ao usar o pipe `%>%` é quebrar a linha após cada pipe para melhorar a legibilidade do código.

3.4 Dados no formato *tidy*

“Tidy datasets are all alike, but every messy dataset is messy in its own way.” —
Hadley Wickham.

Um mesmo conjunto de dados pode ser representado de diversas maneiras. Veja o código abaixo que mostra o mesmo dado em três diferentes formatos.

```
table1
```

```
# A tibble: 6 x 4
  country    year cases population
  <chr>      <dbl> <dbl>      <dbl>
1 Afghanistan 1999    745   19987071
2 Afghanistan 2000   2666   20595360
3 Brazil      1999  37737   172006362
4 Brazil      2000  80488   174504898
5 China       1999 212258  1272915272
6 China       2000 213766  1280428583
```

table2

```
# A tibble: 12 x 4
  country      year type      count
  <chr>      <dbl> <chr>      <dbl>
1 Afghanistan 1999 cases        745
2 Afghanistan 1999 population 19987071
3 Afghanistan 2000 cases        2666
4 Afghanistan 2000 population 20595360
5 Brazil       1999 cases        37737
6 Brazil       1999 population 172006362
7 Brazil       2000 cases        80488
8 Brazil       2000 population 174504898
9 China        1999 cases        212258
10 China       1999 population 1272915272
11 China       2000 cases        213766
12 China       2000 population 1280428583
```

table4a

```
# A tibble: 3 x 3
  country `1999` `2000`
  <chr>      <dbl> <dbl>
1 Afghanistan    745    2666
2 Brazil        37737  80488
3 China         212258 213766
```

Todas as representações acima são dos mesmos dados, mas não são igualmente fáceis de utilizar. A **table1**, por exemplo, será muito mais acessível para trabalhar dentro do tidyverse devido à sua organização no formato *tidy*. Existem três regras inter-relacionadas que caracterizam um conjunto de dados no formato *tidy*:

1. Cada variável é uma coluna; cada coluna representa uma variável.
2. Cada observação é uma linha; cada linha representa uma observação.
3. Cada valor é uma célula; cada célula contém um único valor.

A figura abaixo representa graficamente este conceito.

A pivotação de dados é o processo de reorganizar um conjunto de dados para torná-lo compatível com o formato *tidy*. Isso envolve transformar os dados de um formato mais largo para um

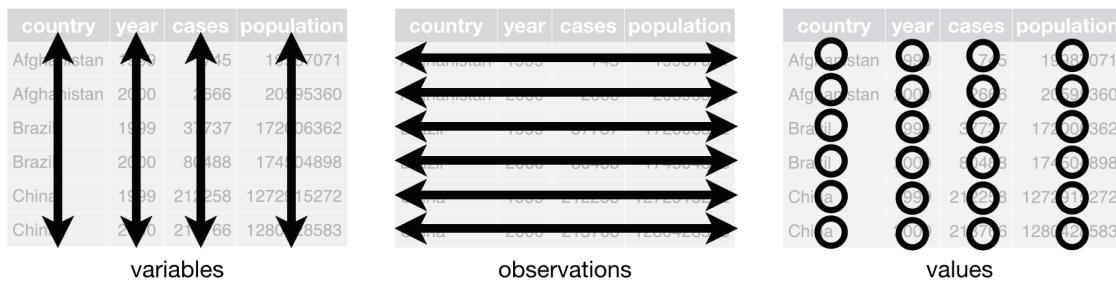


Figura 3.1: Imagem do livro R4DS.

formato mais longo, ou vice-versa, para garantir que cada variável corresponda a uma coluna e cada observação a uma linha.

No exemplo abaixo, estamos transformando os dados de `table2` para um formato mais largo, onde cada valor único da variável `type` se torna uma nova coluna. Note que cada unidade de informação (país, ano, casos e contagem) está quebrado em duas linhas. Então essa operação deixa a tabela de dados mais larga, garantindo que cada unidade de dado esteja representada em uma única linha.

```
table2 %>%
  pivot_wider(names_from="type", values_from="count")
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>      <dbl> <dbl>      <dbl>
1 Afghanistan 1999     745   19987071
2 Afghanistan 2000    2666  20595360
3 Brazil       1999   37737  172006362
4 Brazil       2000   80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

No exemplo abaixo, estamos transformando os dados de `table4a` em um formato mais longo, onde as colunas representando anos específicos (1999 e 2000) são reunidas em uma única coluna chamada `year`, e os valores correspondentes são colocados em uma nova coluna chamada `cases`. Neste caso, a informação sobre os anos estavam armazenadas como nome de colunas, mas, pelo princípio de dados *tidy*, deveriam estar em colunas. Por isso, utilizamos a função `pivot_longer`.


```
table4a %>%
  pivot_longer(cols = c(`1999`, `2000`), names_to = "year", values_to = "cases")
```

```
# A tibble: 6 x 3
  country    year  cases
  <chr>      <chr> <dbl>
1 Afghanistan 1999    745
2 Afghanistan 2000   2666
3 Brazil      1999  37737
4 Brazil      2000  80488
5 China       1999 212258
6 China       2000 213766
```

As duas funções `pivot_wider` e `pivot_longer` são suficientes para fazer a transformação de bases de dados no formato *tidy*.

3.5 Principais verbos do pacote dplyr

O pacote `dplyr` é uma das ferramentas mais poderosas para manipulação de dados no ambiente R. Ele oferece um conjunto coeso de funções que simplificam tarefas comuns de manipulação, como filtragem, seleção, agrupamento, ordenação e resumo de dados. O `dplyr` utiliza uma sintaxe intuitiva e consistente, facilitando a escrita de código limpo e legível.

A seguir, vamos estudar o funcionamento dos principais verbos do pacote. Para exemplificar, vamos utilizar a base de dados `gapminder`. Ela é uma coleção de informações socioeconômicas de diversos países ao longo do tempo, veja Rosling (2012). Ela inclui variáveis como expectativa de vida, PIB per capita, taxa de mortalidade infantil e tamanho da população para diferentes países e anos, cobrindo um período de várias décadas.

Para carregar a base de dados `gapminder`, você precisa carregar o pacote `gapminder`. Com o pacote `gapminder` carregado, a base de dados `gapminder` estará disponível para uso em seu ambiente R:

```
library(gapminder)
```

Warning: package 'gapminder' was built under R version 4.2.3

```
head(gapminder)
```

A função `glimpse()` fornece uma visão geral rápida e concisa da estrutura de um conjunto de dados. Quando aplicada a um conjunto de dados, como o `gapminder`, ela exibe informações essenciais sobre as variáveis presentes, incluindo a quantidade de linhas, colunas e as primeiras linhas do conjunto de dados:

```
glimpse(gapminder)
```

```
Rows: 1,704
```

```
Columns: 6
```

```
$ country <fct> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan", ~
$ continent <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, ~
$ year <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, ~
$ lifeExp <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.854, 40.8~
$ pop <int> 8425333, 9240934, 10267083, 11537966, 13079460, 14880372, 12~
$ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 786.1134, ~
```

3.5.1 select

O verbo `select()` é utilizado para selecionar colunas específicas de um conjunto de dados. Com o `select()`, é possível escolher as colunas desejadas com base em seus nomes, tipos de dados ou outros critérios.

Por exemplo, considerando a base de dados `gapminder`, suponha que desejamos selecionar apenas as colunas referentes ao ano, ao país, à expectativa de vida e ao PIB per capita. Podemos fazer isso da seguinte maneira:

```
# Selecionando as colunas pelo nome
gapminder %>%
  select(year, country, lifeExp, gdpPercap)

# Selecionando apenas as colunas numéricas
gapminder %>%
  select(where(is.numeric))

# Selecionando colunas que começam com "co"
gapminder %>%
  select(starts_with("co"))
```

Note que nos exemplos acima, nenhuma das seleções foi salva em variável. Para salvar as seleções em uma variável, você pode atribuir o resultado de cada operação `select()` a uma variável separada. Por exemplo:

```
gapminder_character <- gapminder %>%
  select(where(is.character))
```

3.5.2 arrange

O verbo `arrange()` é usado para reorganizar as linhas de um conjunto de dados com base nos valores de uma ou mais colunas. Quando aplicado a um conjunto de dados, o `arrange()` classifica as linhas em ordem crescente ou decrescente com base nos valores das colunas especificadas.

No primeiro exemplo usando o verbo `select()`, podemos ordenar os dados por país em ordem alfabética, podemos fazer assim:

```
gapminder %>%
  select(year, country, lifeExp, gdpPercap, pop) %>%
  arrange(country)
```

```
# A tibble: 1,704 x 5
   year country    lifeExp gdpPercap    pop
  <int> <fct>      <dbl>    <dbl>    <int>
1  1952 Afghanistan  28.8      779.  8425333
2  1957 Afghanistan  30.3      821.  9240934
3  1962 Afghanistan  32.0      853. 10267083
4  1967 Afghanistan  34.0      836. 11537966
5  1972 Afghanistan  36.1      740. 13079460
6  1977 Afghanistan  38.4      786. 14880372
7  1982 Afghanistan  39.9      978. 12881816
8  1987 Afghanistan  40.8      852. 13867957
9  1992 Afghanistan  41.7      649. 16317921
10 1997 Afghanistan  41.8      635. 22227415
# i 1,694 more rows
```

No exemplo abaixo, estamos organizando de acordo com o ano em ordem crescente e a expectativa de vida em ordem decrescente dentro de cada ano.

```
gapminder %>%
  select(year, country, lifeExp, gdpPercap, pop) %>%
  arrange(year, desc(lifeExp))
```

```
# A tibble: 1,704 x 5
  year country      lifeExp gdpPercap      pop
  <int> <fct>      <dbl>      <dbl>    <int>
1  1952 Norway      72.7    10095.  3327728
2  1952 Iceland     72.5     7268.   147962
3  1952 Netherlands  72.1     8942. 10381988
4  1952 Sweden      71.9     8528.  7124673
5  1952 Denmark     70.8     9692.  4334000
6  1952 Switzerland  69.6    14734.  4815000
7  1952 New Zealand  69.4    10557.  1994794
8  1952 United Kingdom 69.2     9980. 50430000
9  1952 Australia    69.1    10040.  8691212
10 1952 Canada      68.8    11367. 14785584
# i 1,694 more rows
```

Dica

Ao utilizar o verbo `select()` com o prefixo `-`, você pode especificar as colunas que deseja **excluir** do conjunto de dados. No exemplo abaixo, vamos excluir a coluna `continent` da seleção no conjunto de dados.

```
gapminder %>%
  select(-continent)
```

```
# A tibble: 1,704 x 5
  country      year lifeExp      pop gdpPercap
  <fct>      <int>   <dbl>    <int>    <dbl>
1 Afghanistan 1952    28.8  8425333    779.
2 Afghanistan 1957    30.3  9240934    821.
3 Afghanistan 1962    32.0 10267083    853.
4 Afghanistan 1967    34.0 11537966    836.
5 Afghanistan 1972    36.1 13079460    740.
6 Afghanistan 1977    38.4 14880372    786.
7 Afghanistan 1982    39.9 12881816    978.
8 Afghanistan 1987    40.8 13867957    852.
9 Afghanistan 1992    41.7 16317921    649.
10 Afghanistan 1997    41.8 22227415    635.
# i 1,694 more rows
```

3.5.3 filter

Para analisar dados específicos de interesse, muitas vezes é necessário filtrar o conjunto de dados para incluir apenas as observações relevantes. O verbo `filter()` é usado para fazer isso. Basta definir uma ou mais condições lógicas que as linhas da base de dados devem satisfazer para serem mostradas.

No exemplo abaixo, estamos filtrando os dados para incluir apenas as observações onde o país é “Brasil” ou “Argentina”.

```
gapminder %>%
  select(year, country, lifeExp, gdpPercap, pop) %>%
  arrange(year, desc(lifeExp)) %>%
  filter(country == "Brazil" | country == "Argentina")
```

```
# A tibble: 24 x 5
   year country  lifeExp gdpPercap      pop
  <int> <fct>      <dbl>    <dbl>    <int>
1  1952 Argentina   62.5    5911.  17876956
2  1952 Brazil     50.9    2109.  56602560
3  1957 Argentina   64.4    6857.  19610538
4  1957 Brazil     53.3    2487.  65551171
5  1962 Argentina   65.1    7133.  21283783
6  1962 Brazil     55.7    3337.  76039390
7  1967 Argentina   65.6    8053.  22934225
8  1967 Brazil     57.6    3430.  88049823
9  1972 Argentina   67.1    9443.  24779799
10 1972 Brazil     59.5    4986. 100840058
# i 14 more rows
```

3.5.4 mutate

O verbo `mutate()` é usado para criar ou modificar colunas em um conjunto de dados existente. Ele permite adicionar novas variáveis calculadas com base em variáveis existentes ou modificar as variáveis existentes de acordo com alguma lógica específica.

Por exemplo, podemos usar o `mutate()` para calcular uma nova variável que represente o PIB total de cada país multiplicando o PIB per capita pelo tamanho da população. Aqui está um exemplo de como fazer isso com o conjunto de dados `gapminder`:

```
gapminder_total_gdp <- gapminder %>%
  select(country, year, lifeExp, gdpPercap, pop) %>%
  mutate(total_gdp = gdpPercap * pop)
```

3.5.5 summarise

O verbo `summarise()` é usado para resumir os dados em uma única linha, geralmente calculando estatísticas resumidas como média, soma, mediana, etc. Ele permite calcular resumos estatísticos em um conjunto de dados, criando uma nova tabela contendo os resultados resumidos.

Aqui está um exemplo de como usar `summarise()` para calcular a média da expectativa de vida usando os dados do `gapminder`:

```
gapminder %>%
  summarise(mean_lifeExp = mean(lifeExp, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  mean_lifeExp
    <dbl>
1      59.5
```

3.5.6 group by

O verbo `group_by()` é usado para dividir os dados em grupos com base nos valores de uma ou mais variáveis. Ele não realiza cálculos por si só, mas altera o comportamento das funções de resumo, como `summarise()`, para operar em cada grupo separadamente.

Aqui está um exemplo de como usar `group_by()` com os dados do `gapminder` para calcular a média da expectativa de vida por continente:

```
gapminder %>%
  group_by(continent) %>%
  summarise(mean_lifeExp = mean(lifeExp, na.rm = TRUE))
```

```
# A tibble: 5 x 2
  continent mean_lifeExp
    <fct>         <dbl>
1 Africa         48.9
2 Americas       64.7
```

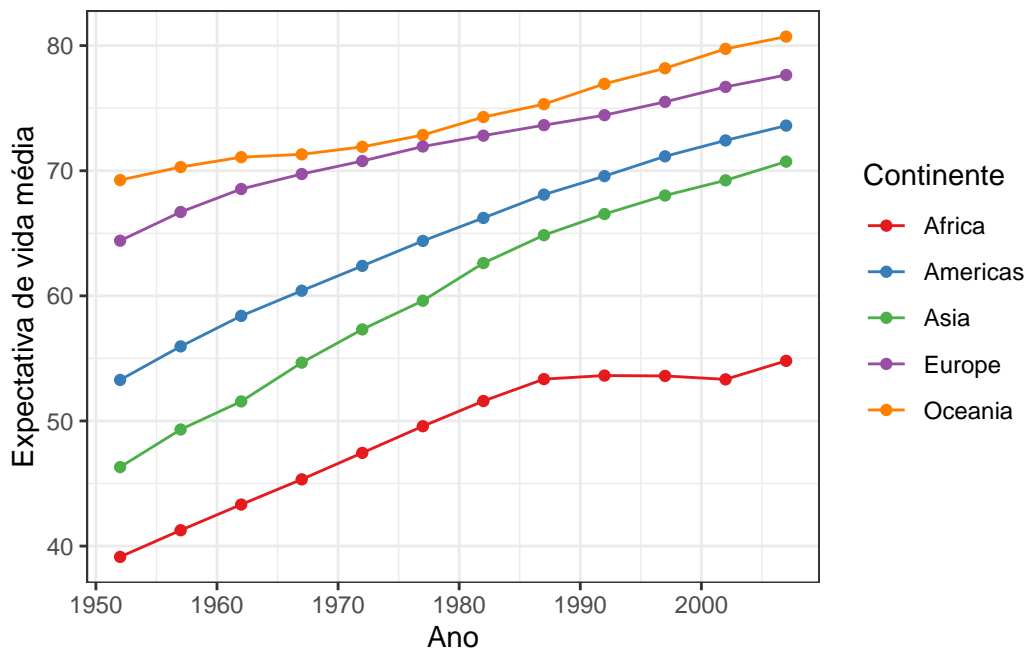
3 Asia	60.1
4 Europe	71.9
5 Oceania	74.3

O exemplo abaixo utiliza todos os principais verbos do `dplyr` para calcular a expectativa de vida média e o PIB (em milhares) médio por continente no ano de 2007.

```
gapminder %>%
  select(country, continent, year, lifeExp, gdpPercap) %>%
  filter(year == 2007) %>% # apenas os dados para o ano de 2007
  mutate(gdp = gdpPercap / 1000) %>% # representa o PIB per capita em milhares
  group_by(continent) %>% # agrupar os dados por continente
  summarise(mean_lifeExp = mean(lifeExp, na.rm = TRUE), # média da expectativa de vida
            mean_gdp = mean(gdp, na.rm = TRUE)) %>% #média do PIB per capita em bilhões
  arrange(desc(mean_lifeExp))
```

```
# A tibble: 5 x 3
  continent mean_lifeExp mean_gdp
  <fct>      <dbl>      <dbl>
1 Oceania    80.7        29.8
2 Europe     77.6        25.1
3 Americas   73.6        11.0
4 Asia       70.7        12.5
5 Africa     54.8         3.09
```

O gráfico abaixo mostra a evolução da expectativa de vida média nos continentes ao longo dos anos.



💡 Desafio

Qual mudança foi feita no código do exemplo anterior para construir os dados usados na geração deste gráfico?

3.6 Funções auxiliares

Apresentando funções auxiliares do pacote `dplyr` que podem ser muito úteis em diversos contextos.

- `pull`, `distinct`, `unite`, `separate_wider_delim`, e a família de funções `slice_*`.

```
gapminder %>%
  filter(year == 1952) %>%
  pull(continent)
```

```
[1] Asia      Europe    Africa    Africa    Americas  Oceania    Europe    Asia
[9] Asia      Europe    Africa    Americas  Europe    Africa    Americas  Europe
[17] Africa    Africa    Asia      Africa    Americas  Africa    Africa    Americas
[25] Asia      Americas  Africa    Africa    Africa    Americas  Africa    Europe
[33] Americas  Europe    Europe    Africa    Americas  Americas  Africa    Americas
```



```

[41] Africa Africa Africa Europe Europe Africa Africa Europe
[49] Africa Europe Americas Africa Africa Americas Americas Asia
[57] Europe Europe Asia Asia Asia Asia Europe Asia
[65] Europe Americas Asia Asia Africa Asia Asia Asia
[73] Asia Africa Africa Africa Africa Africa Asia Africa
[81] Africa Africa Americas Asia Europe Africa Africa Asia
[89] Africa Asia Europe Oceania Americas Africa Africa Europe
[97] Asia Asia Americas Americas Americas Asia Europe Europe
[105] Americas Africa Europe Africa Africa Asia Africa Europe
[113] Africa Asia Europe Europe Africa Africa Europe Asia
[121] Africa Africa Europe Europe Asia Asia Africa Asia
[129] Africa Americas Africa Europe Africa Europe Americas Americas
[137] Americas Asia Asia Asia Africa Africa
Levels: Africa Americas Asia Europe Oceania

```

```

gapminder %>%
  distinct(continent)

```

```

# A tibble: 5 x 1
  continent
  <fct>
1 Asia
2 Europe
3 Africa
4 Americas
5 Oceania

```

```

gapminder %>%
  slice(1:10)

```

```

# A tibble: 10 x 6
  country      continent  year lifeExp      pop gdpPercap
  <fct>        <fct>    <int> <dbl>    <int>    <dbl>
1 Afghanistan Asia      1952  28.8  8425333    779.
2 Afghanistan Asia      1957  30.3  9240934    821.
3 Afghanistan Asia      1962  32.0 10267083    853.
4 Afghanistan Asia      1967  34.0 11537966    836.
5 Afghanistan Asia      1972  36.1 13079460    740.
6 Afghanistan Asia      1977  38.4 14880372    786.
7 Afghanistan Asia      1982  39.9 12881816    978.

```

8	Afghanistan	Asia	1987	40.8	13867957	852.
9	Afghanistan	Asia	1992	41.7	16317921	649.
10	Afghanistan	Asia	1997	41.8	22227415	635.

```
gapminder %>%
  slice_head(n = 5)
```

```
# A tibble: 5 x 6
  country      continent year lifeExp      pop gdpPercap
  <fct>        <fct>    <int>   <dbl>   <int>    <dbl>
1 Afghanistan Asia      1952    28.8  8425333    779.
2 Afghanistan Asia      1957    30.3  9240934    821.
3 Afghanistan Asia      1962    32.0 10267083    853.
4 Afghanistan Asia      1967    34.0 11537966    836.
5 Afghanistan Asia      1972    36.1 13079460    740.
```

```
gapminder %>%
  slice_tail(n = 5)
```

```
# A tibble: 5 x 6
  country      continent year lifeExp      pop gdpPercap
  <fct>        <fct>    <int>   <dbl>   <int>    <dbl>
1 Zimbabwe Africa      1987    62.4  9216418    706.
2 Zimbabwe Africa      1992    60.4 10704340    693.
3 Zimbabwe Africa      1997    46.8 11404948    792.
4 Zimbabwe Africa      2002    40.0 11926563    672.
5 Zimbabwe Africa      2007    43.5 12311143    470.
```

```
set.seed(1)
gapminder %>%
  slice_sample(n = 10)
```

```
# A tibble: 10 x 6
  country      continent year lifeExp      pop gdpPercap
  <fct>        <fct>    <int>   <dbl>   <int>    <dbl>
1 Montenegro Europe      1992    75.4   621621   7003.
2 Hungary     Europe      1982    69.4 10705535 12546.
3 Benin       Africa      1992    53.9  4981671  1191.
```

4	Malawi	Africa	1977	43.8	5637246	663.
5	Thailand	Asia	1992	67.3	56667095	4617.
6	El Salvador	Americas	1962	52.3	2747687	3777.
7	China	Asia	2002	72.0	1280400000	3119.
8	Chad	Africa	1977	47.4	4388260	1134.
9	Peru	Americas	2002	69.9	26769436	5909.
10	Senegal	Africa	2002	61.6	10870037	1520.

```
gapminder %>%
  filter(year == 2007) %>%
  slice_max(lifeExp, n = 2)
```

A tibble: 2 x 6

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	Japan	Asia	2007	82.6	127467972	31656.
2	Hong Kong, China	Asia	2007	82.2	6980412	39725.

```
gapminder %>%
  filter(year == 2007) %>%
  slice_min(lifeExp, n = 2)
```

A tibble: 2 x 6

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	Swaziland	Africa	2007	39.6	1133066	4513.
2	Mozambique	Africa	2007	42.1	19951656	824.

```
gapminder %>%
  filter(year == 2007 | year == 1952) %>%
  group_by(year) %>%
  slice_max(lifeExp, n = 2)
```

A tibble: 4 x 6

Groups: year [2]

	country	continent	year	lifeExp	pop	gdpPercap
	<fct>	<fct>	<int>	<dbl>	<int>	<dbl>
1	Norway	Europe	1952	72.7	3327728	10095.

2	Iceland	Europe	1952	72.5	147962	7268.
3	Japan	Asia	2007	82.6	127467972	31656.
4	Hong Kong, China	Asia	2007	82.2	6980412	39725.

```
gapminder_united <- gapminder %>%
  unite("country_continent", c(country, continent),
        sep = "_",
        remove = TRUE,
        na.rm = FALSE)

gapminder_united %>%
  separate_wider_delim(country_continent,
                        delim = "_",
                        names = c("country", "continent"))
```

```
# A tibble: 1,704 x 6
  country      continent year lifeExp      pop gdpPercap
  <chr>        <chr>    <int>  <dbl>    <int>    <dbl>
1 Afghanistan Asia      1952   28.8  8425333    779.
2 Afghanistan Asia      1957   30.3  9240934    821.
3 Afghanistan Asia      1962   32.0 10267083    853.
4 Afghanistan Asia      1967   34.0 11537966    836.
5 Afghanistan Asia      1972   36.1 13079460    740.
6 Afghanistan Asia      1977   38.4 14880372    786.
7 Afghanistan Asia      1982   39.9 12881816    978.
8 Afghanistan Asia      1987   40.8 13867957    852.
9 Afghanistan Asia      1992   41.7 16317921    649.
10 Afghanistan Asia      1997   41.8 22227415    635.
# i 1,694 more rows
```

3.7 Exercícios

Vamos trabalhar com o conjunto de dados `billboard`. Neste conjunto de dados, cada observação é uma música. As três primeiras colunas (artista, faixa e data de entrada) são variáveis que descrevem a música. Em seguida, temos 76 colunas (`wk1-wk76`) que descrevem o ranking da música em cada semana. Aqui, os nomes das colunas são uma variável (a semana) e os valores das células são outra (o ranking).

```
library(tidyverse)
billboard
```

```
# A tibble: 317 x 79
  artist      track date.entered wk1 wk2 wk3 wk4 wk5 wk6 wk7 wk8
  <chr>      <chr> <date>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 2 Pac      Baby~ 2000-02-26      87  82  72  77  87  94  99  NA
2 2Ge+her    The ~ 2000-09-02      91  87  92  NA  NA  NA  NA  NA
3 3 Doors D~ Kryp~ 2000-04-08      81  70  68  67  66  57  54  53
4 3 Doors D~ Loser 2000-10-21      76  76  72  69  67  65  55  59
5 504 Boyz   Wobb~ 2000-04-15      57  34  25  17  17  31  36  49
6 98^0       Give~ 2000-08-19      51  39  34  26  26  19   2   2
7 A*Teens    Danc~ 2000-07-08      97  97  96  95 100  NA  NA  NA
8 Aaliyah    I Do~ 2000-01-29      84  62  51  41  38  35  35  38
9 Aaliyah    Try ~ 2000-03-18      59  53  38  28  21  18  16  14
10 Adams, Yo~ Open~ 2000-08-26      76  76  74  69  68  67  61  58
# i 307 more rows
# i 68 more variables: wk9 <dbl>, wk10 <dbl>, wk11 <dbl>, wk12 <dbl>,
# wk13 <dbl>, wk14 <dbl>, wk15 <dbl>, wk16 <dbl>, wk17 <dbl>, wk18 <dbl>,
# wk19 <dbl>, wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>,
# wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>, wk30 <dbl>,
# wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>, wk35 <dbl>, wk36 <dbl>,
# wk37 <dbl>, wk38 <dbl>, wk39 <dbl>, wk40 <dbl>, wk41 <dbl>, wk42 <dbl>, ...
```

a) Aplique uma transformação na base de dados para deixá-la no formato abaixo.

```
# A tibble: 24,092 x 5
  artist track      date.entered week  rank
  <chr>  <chr>      <date>      <chr> <dbl>
1 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk1    87
2 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk2    82
3 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk3    72
4 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk4    77
5 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk5    87
6 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk6    94
7 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk7    99
8 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk8    NA
9 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk9    NA
10 2 Pac  Baby Don't Cry (Keep... 2000-02-26 wk10   NA
# i 24,082 more rows
```

b) Observe o resultado do item a). O que acontece se uma música estiver no top 100 por menos de 76 semanas? Pegue a música “Baby Don’t Cry” de 2 Pac, por exemplo. A saída acima sugere que ela esteve no top 100 por apenas 7 semanas, e todas as semanas restantes são preenchidas com valores ausentes (NA). Esses NAs na verdade não representam observações

desconhecidas; eles foram forçados a existir pela estrutura do conjunto de dados. Altere o código usado em **a)** para remover esses NAs. Responda: Quantas linhas sobraram? (Dica: veja a documentação da função `pivot_longer`.)

c) Você deve ter percebido que no resultado do item **a)**, o tipo da coluna `week` é caractere. Faça a transformação adequada para obter uma coluna com valores numéricos.

d) Qual música ficou por mais semanas no top 100 da Billboard em 2000? Por quantas semanas essa música apareceu no ranking? E qual música ficou por menos tempo no ranking.

e) Qual música ficou exatamente 10 semanas no top 100 da Billboard em 2000? Caso exista mais de uma música nessa condição, considere a que primeiro entrou no ranking.

4 Visualização de dados

A visualização de dados desempenha um papel fundamental na compreensão e comunicação de informações complexas. Ao transformar números e estatísticas em gráficos, tabelas e outras representações visuais, podemos identificar padrões, tendências e insights que podem não ser imediatamente aparentes em conjuntos de dados brutos. Além disso, a visualização de dados facilita a comunicação de resultados e descobertas para uma audiência mais ampla, tornando informações complexas mais acessíveis e compreensíveis.

4.1 Gramática dos Gráficos

Ele é baseado na gramática dos gráficos (*grammar of graphics* - GG), veja Wilkinson (2012). A gramática dos gráficos é um conjunto de princípios e conceitos que descrevem a estrutura e as regras para criar gráficos de forma consistente e eficaz. A GG é um *framework* para visualização de dados que desmembra cada componente de um gráfico em elementos individuais, criando camadas distintas. Usando o sistema GG, podemos construir gráficos passo a passo para obter resultados flexíveis e personalizáveis. Cada aspecto do gráfico, como pontos, linhas, cores e escalas, é tratado como uma peça separada, permitindo um controle detalhado sobre a aparência e o conteúdo do gráfico final.

4.2 O pacote ggplot

O pacote mais conhecido de visualização de dados em R é o **ggplot2**, que é baseado na gramática dos gráficos. O **ggplot2** permite criar uma ampla variedade de gráficos, incluindo gráficos de dispersão, linhas, barras, histogramas, entre outros, de forma simples e flexível. Com o **ggplot2**, você pode personalizar praticamente todos os aspectos do gráfico, desde a forma e a cor dos pontos até a escala dos eixos e a aparência do plano de fundo.

Para exemplificar o uso do **ggplot**, vamos considerar os dados **gapminder**, Rosling (2012).

4.2.1 Dados

A função `ggplot()` inicializa um gráfico **ggplot2** e define os dados que serão usados.

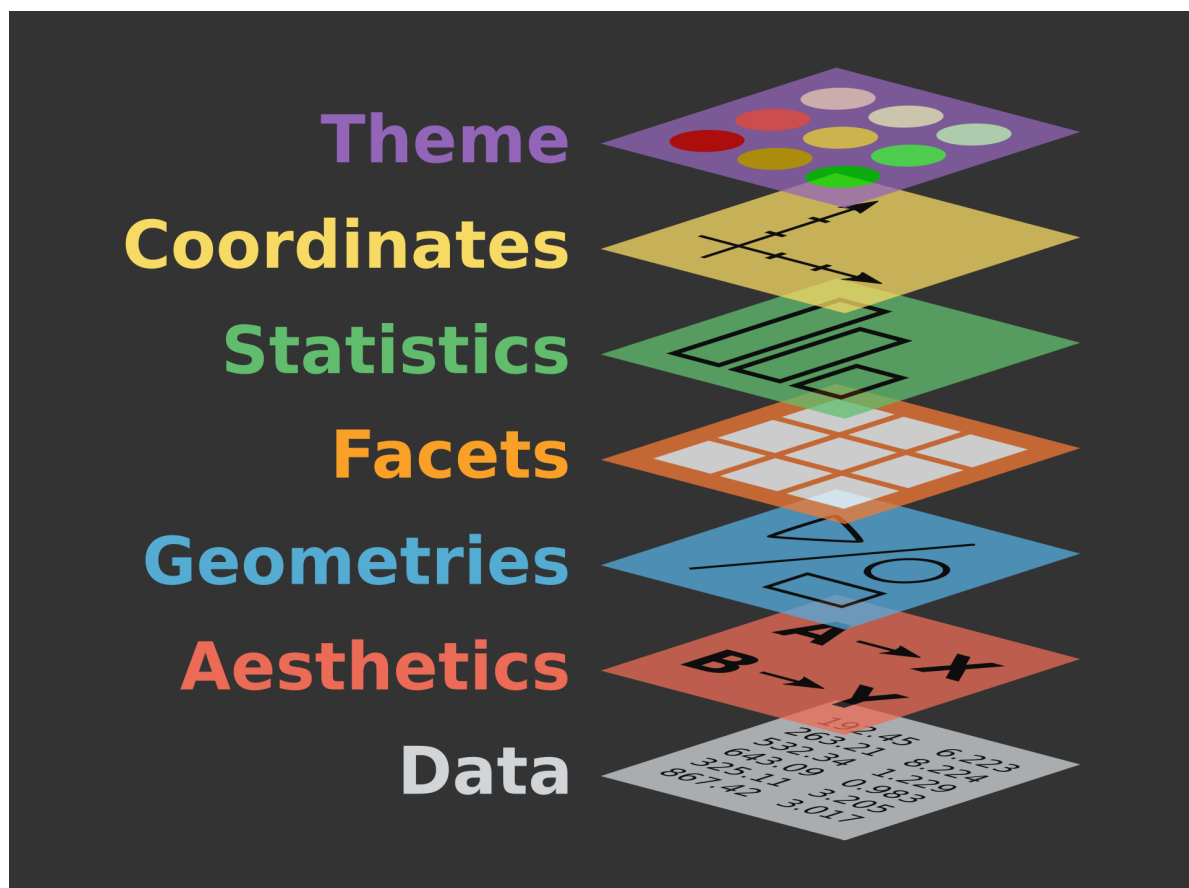
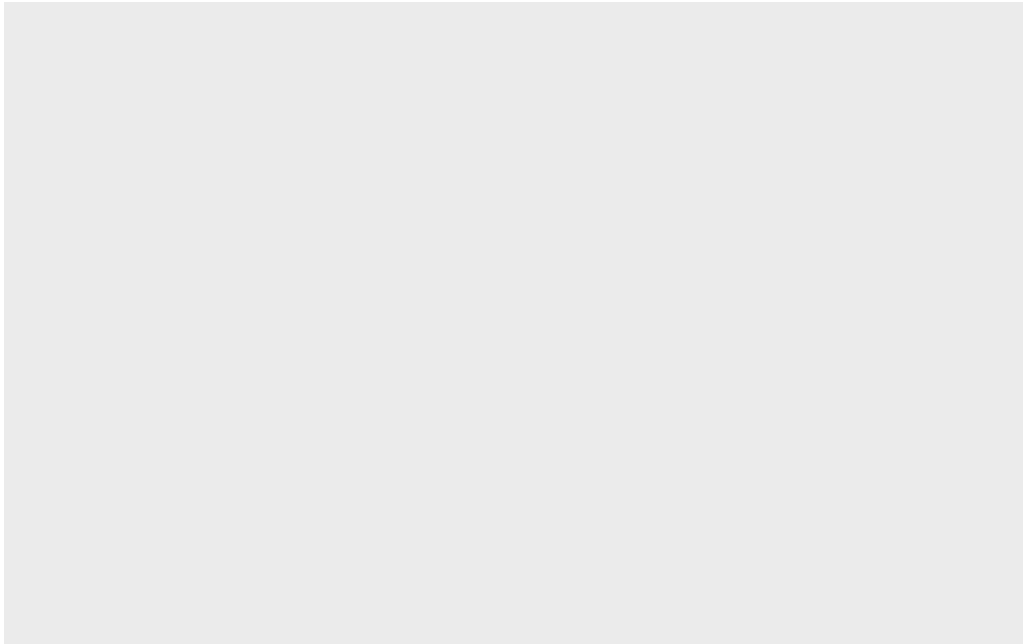


Figura 4.1: Figura de Wilkinson (2012).


```
library(tidyverse)
library(gapminder)

gapminder_2007 <- gapminder %>%
  filter(year == 2007)

ggplot(data = gapminder_2007)
```



i O gráfico gerado também não está errado!

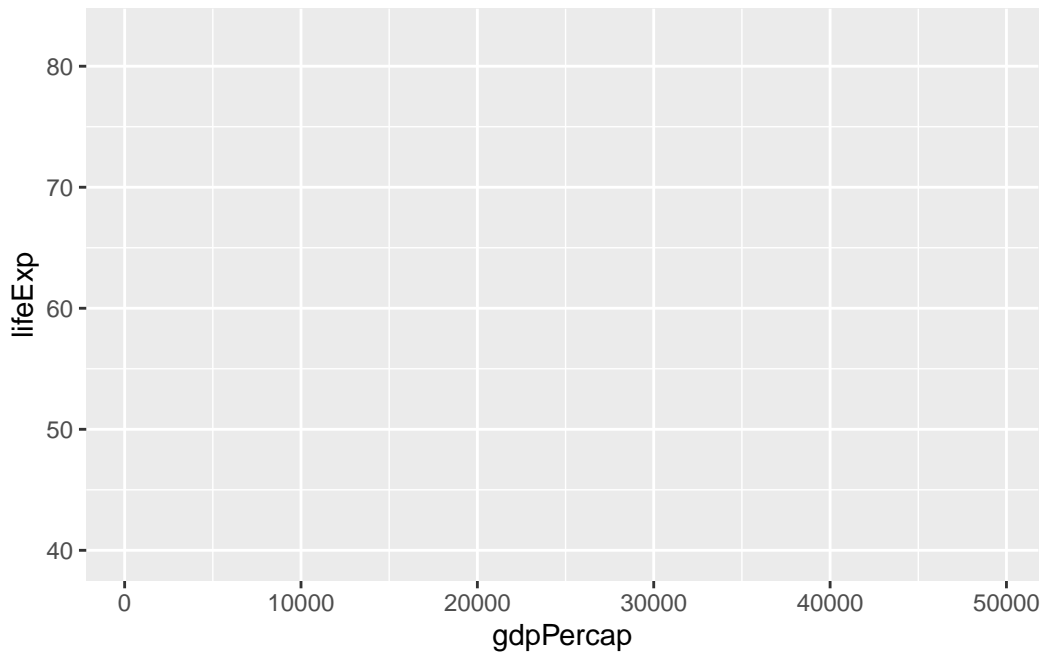
Esse código cria um gráfico utilizando os dados do `gapminder` apenas para o ano de 2007. Apenas isso. Não há instruções extras sobre o que exibir em cada eixo (estética do gráfico).

4.2.2 Estética

O mapeamento estético (`aes`) no `ggplot2` é uma função que permite vincular variáveis de um conjunto de dados às propriedades visuais de um gráfico, como cor, forma, tamanho e posição. Por meio do mapeamento estético, podemos controlar como os dados são representados visualmente no gráfico.

Por exemplo, ao criar um gráfico de dispersão, podemos mapear a variável x (horizontal) e a variável y (vertical) do conjunto de dados às coordenadas do gráfico.

```
ggplot(data = gapminder_2007,  
       mapping = aes(x = gdpPercap, y = lifeExp))
```



i O gráfico gerado também não está errado!

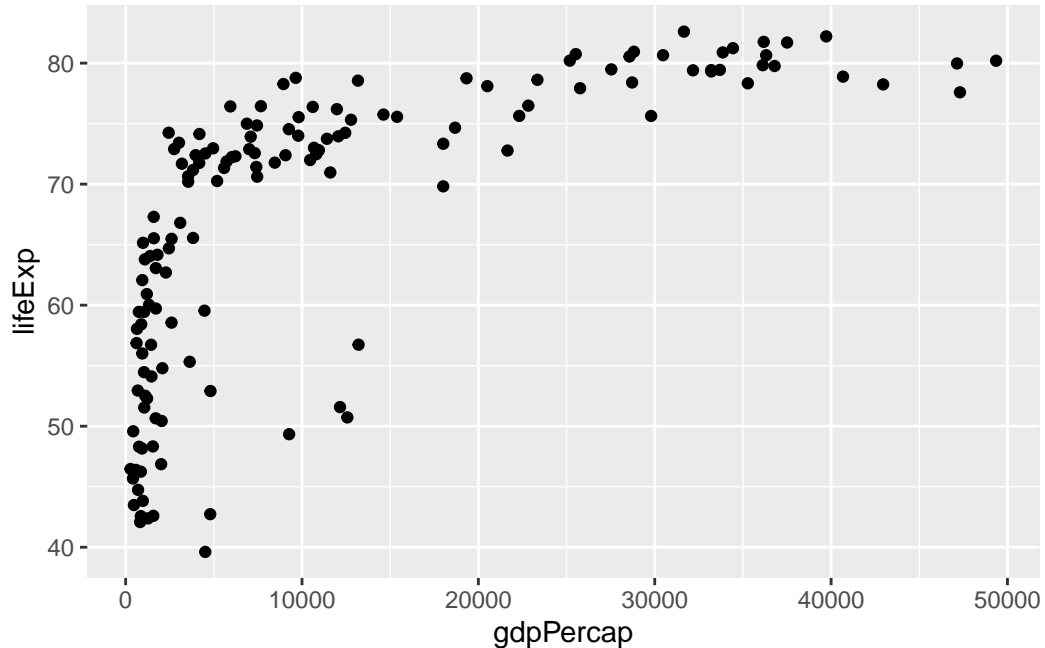
Esse código define um gráfico de dispersão utilizando os dados do gapminder apenas para o ano de 2007. O eixo x representa o PIB per capita (gdpPercap) e o eixo y representa a expectativa de vida (lifeExp). Apenas isso. Não há instruções extra no trecho de código sobre o formato (geometria) que deve ser usado para exibir os dados.

4.2.3 Geometria

A geometria refere-se aos elementos visuais que compõem um gráfico, como pontos, linhas, barras e áreas. Cada tipo de gráfico tem sua própria geometria correspondente, e é especificada pela função `geom_*`, seguida pelo tipo de geometria desejado.

Por exemplo, para criar um gráfico de dispersão, usamos a geometria `geom_point()`, enquanto para criar um gráfico de barras, usamos a geometria `geom_bar()`.

```
ggplot(data = gapminder_2007,
       mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```

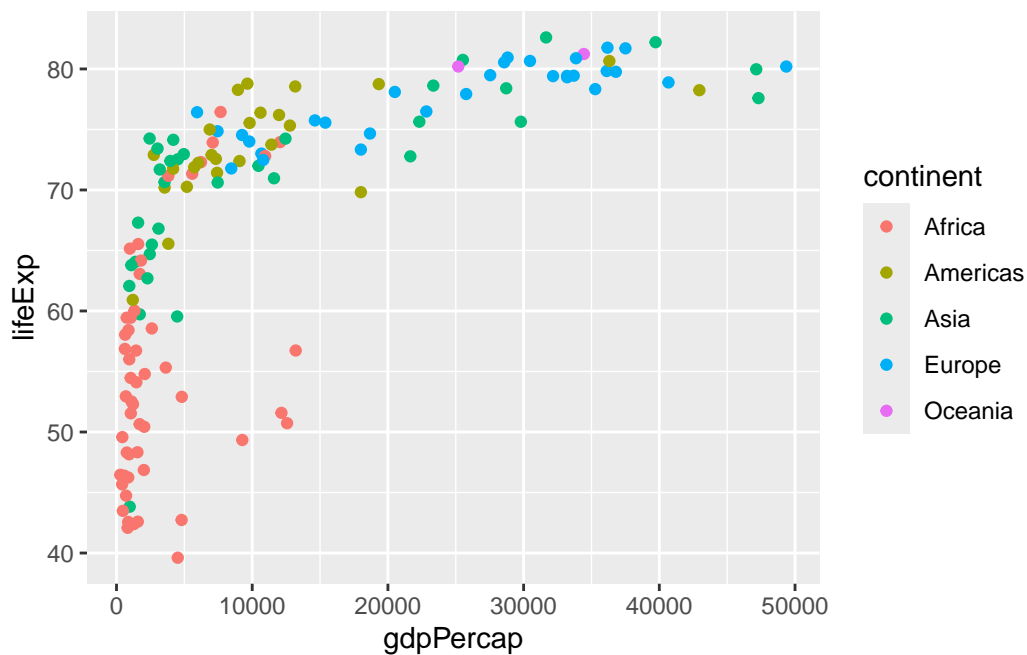


Cada geometria tem seus próprios parâmetros específicos que podem ser ajustados para personalizar a aparência do gráfico, como cor, tamanho, preenchimento e transparência.

💡 Adicionando uma estética de cores no gráfico

Veja que para colorir cada ponto do gráfico de dispersão de acordo com o continente, basta adicionar no mapeamento estético `color = continent`.

```
ggplot(data = gapminder_2007,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point()
```



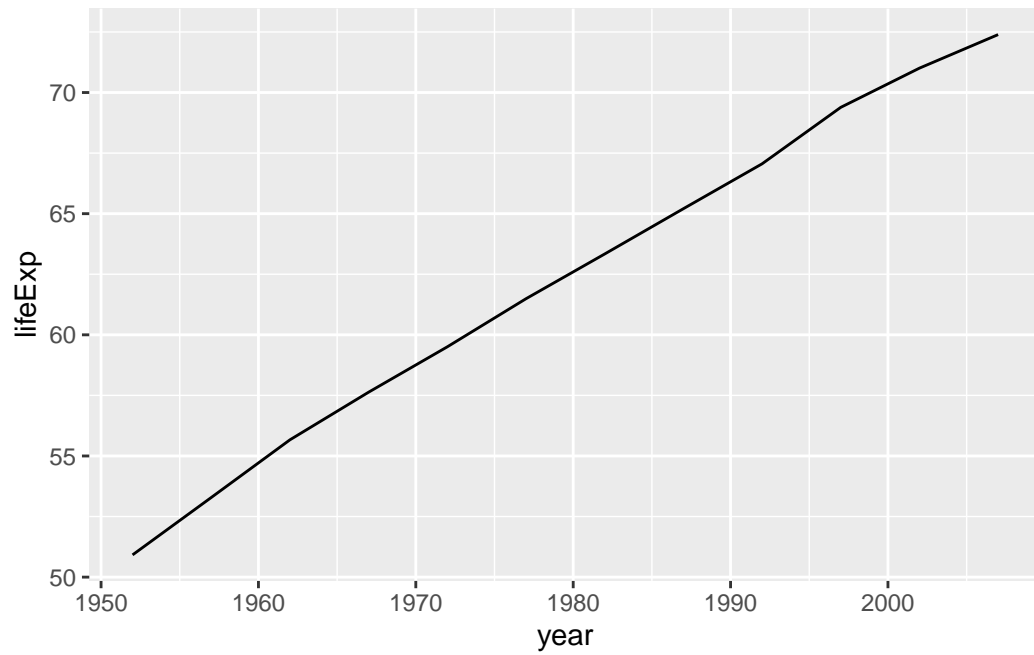
4.2.3.1 Outras geometrias

Abaixo, mostramos outras possíveis geometrias. Algumas alterações foram feitas na forma como as funções do `ggplot` são usadas. Propositamente não foi adicionado texto descritivo pois essas alterações devem ser fáceis de interpretar.

`geom_line()`:

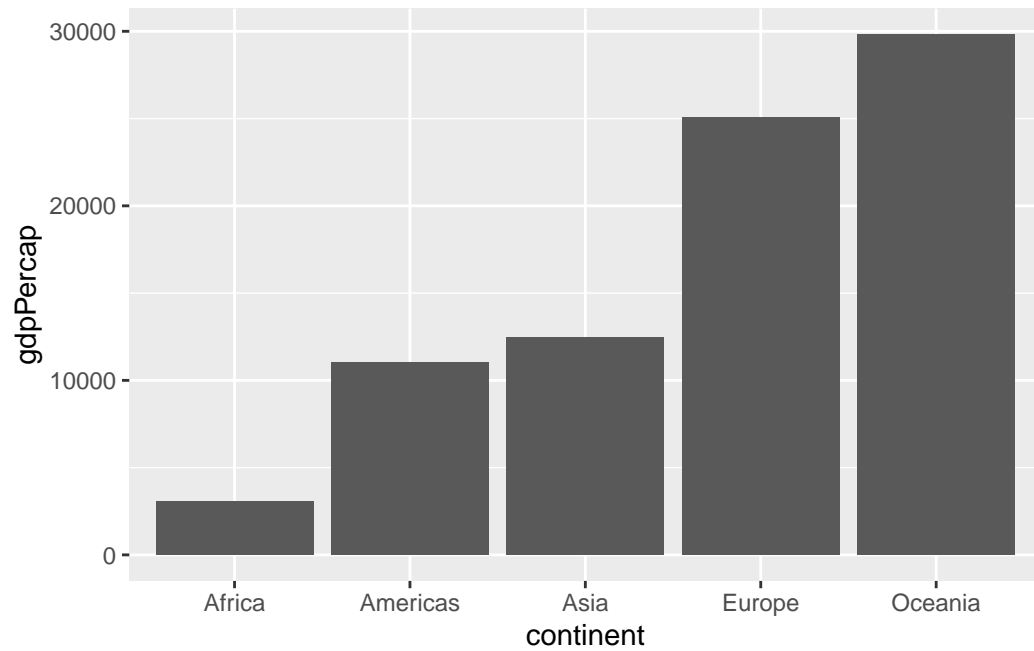
```
# Filtrar dados para o Brasil
dados_pais <- gapminder %>%
  filter(country == "Brazil")

# Criar gráfico de linha
ggplot(data = gapminder %>%
  filter(country == "Brazil"),
  aes(x = year, y = lifeExp)) +
  geom_line()
```



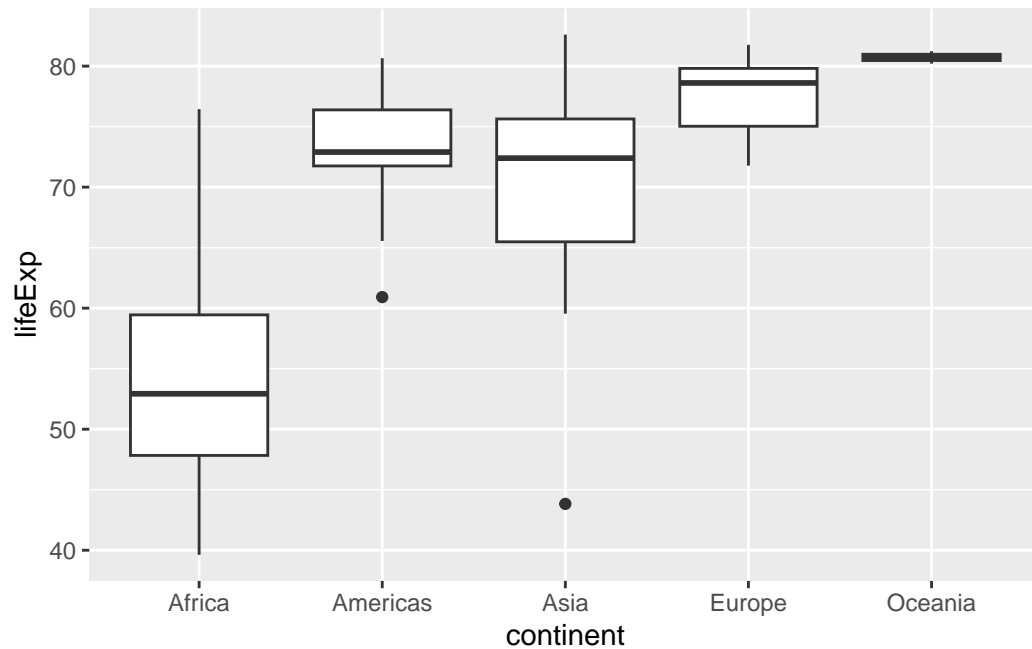
`geom_bar()`:

```
# Criar gráfico de barras
ggplot(data = gapminder_2007, aes(x = continent, y = gdpPercap)) +
  geom_bar(stat = "summary", fun = "mean")
```



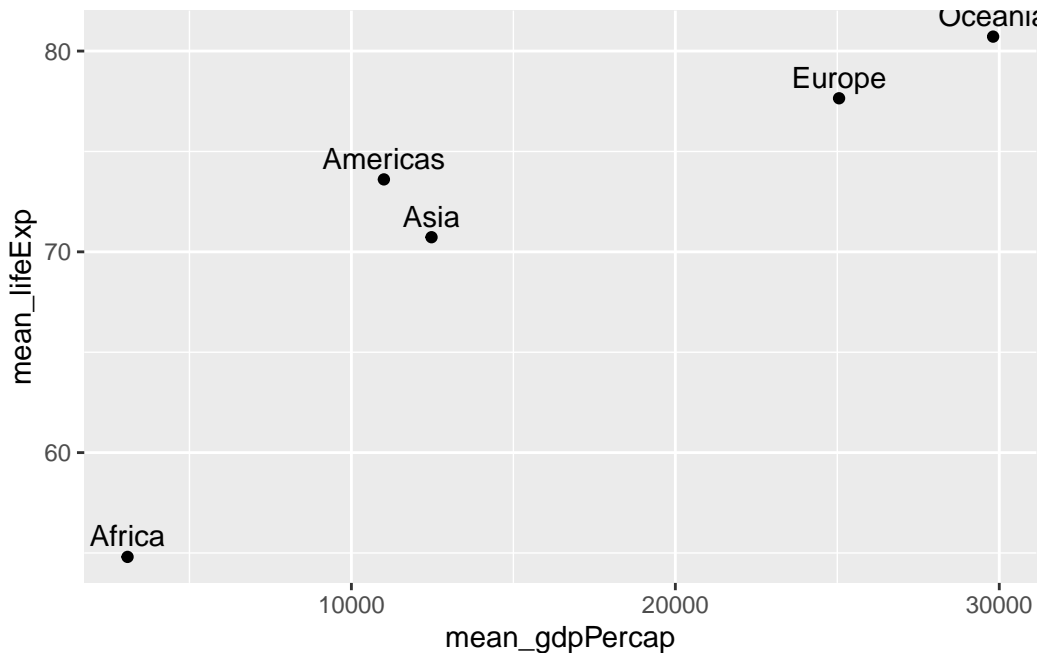
`geom_boxplot():`

```
ggplot(data = gapminder_2007, aes(x = continent, y = lifeExp)) +  
  geom_boxplot()
```



geom_text():

```
gapminder_2007 %>%
  group_by(continent) %>%
  summarise(mean_lifeExp = mean(lifeExp),
            mean_gdpPercap = mean(gdpPercap)) %>%
  ggplot(aes(x = mean_gdpPercap, y = mean_lifeExp, label = continent)) +
  geom_point() +
  geom_text(vjust = -0.5, hjust = 0.5)
```



Note que estamos combinando duas geometrias em um único gráfico (ponto e texto). Os rótulos de texto são adicionados aos pontos usando a geometria `geom_text()`, com os parâmetros `vjust` e `hjust` definindo a posição vertical e horizontal do texto, respectivamente:

- o parâmetro `vjust` ajusta o alinhamento vertical do texto em relação ao ponto. Um valor negativo (-0.5, por exemplo) move o texto acima do ponto, enquanto um valor positivo move o texto abaixo do ponto.
- o parâmetro `hjust` ajusta o alinhamento horizontal do texto em relação ao ponto. Um valor de 0.5 centraliza o texto horizontalmente em relação ao ponto.

4.2.4 Facetas

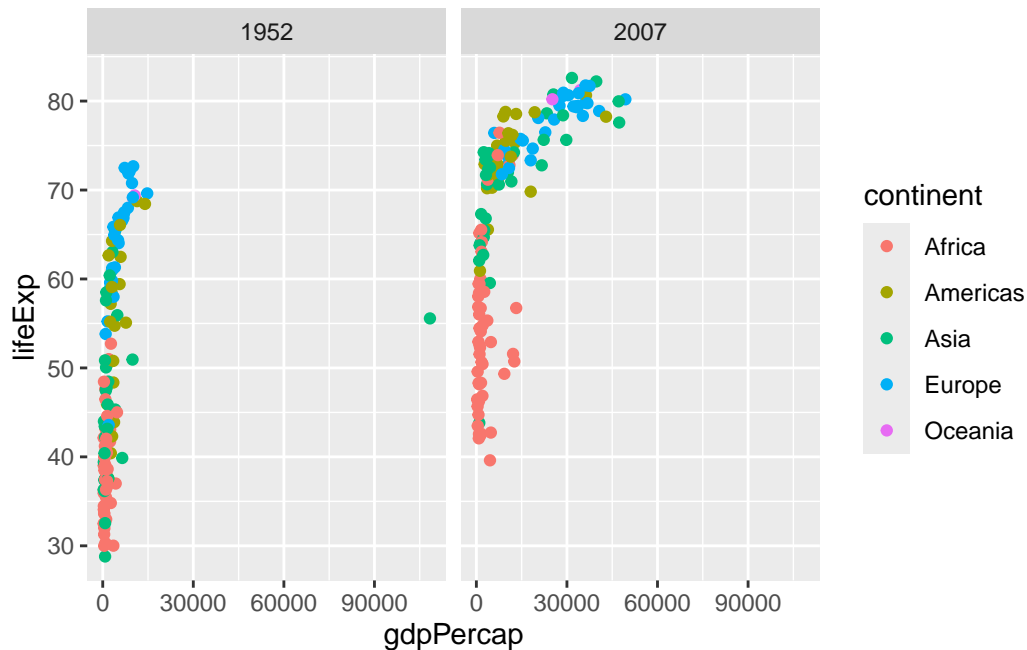
As facetas (`facet`) referem-se à capacidade de dividir um gráfico em múltiplas visualizações com base em uma ou mais variáveis categóricas. Isso permite comparar as relações entre variáveis em diferentes segmentos dos dados.

As facetas são adicionadas usando a função `facet_wrap()` para criar uma matriz de painéis com base em uma variável categórica ou `facet_grid()` para criar uma grade de painéis com base em duas variáveis categóricas.

Por exemplo, podemos usar facetas para criar um gráfico de dispersão separado para dois anos distintos, permitindo comparar as relações entre o PIB per capita e a expectativa de vida nesses dois anos.


```
gapminder_anos <- gapminder %>%
  filter(year == 1952 | year == 2007)

ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  facet_wrap(~year)
```



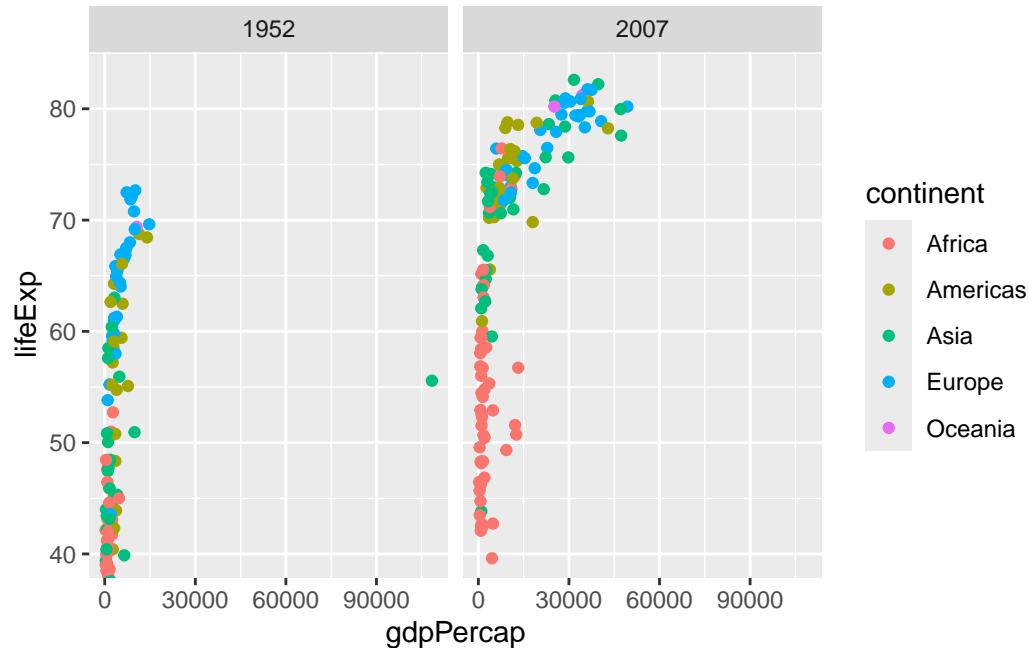
4.2.5 Coordenadas

As coordenadas em ggplot2 determinam como os dados são mapeados em um espaço gráfico. Isso inclui a escala dos eixos x e y, bem como qualquer transformação ou ajuste aplicado aos dados. As coordenadas afetam a aparência geral do gráfico, incluindo sua orientação, proporção e escala.

Para definir limites nos eixos x e y, podemos usar a função `coord_cartesian()` para controlar quais intervalos de valores são exibidos no gráfico. Isso é útil quando queremos focar em uma parte específica dos dados ou evitar que outliers influenciem a escala dos eixos.

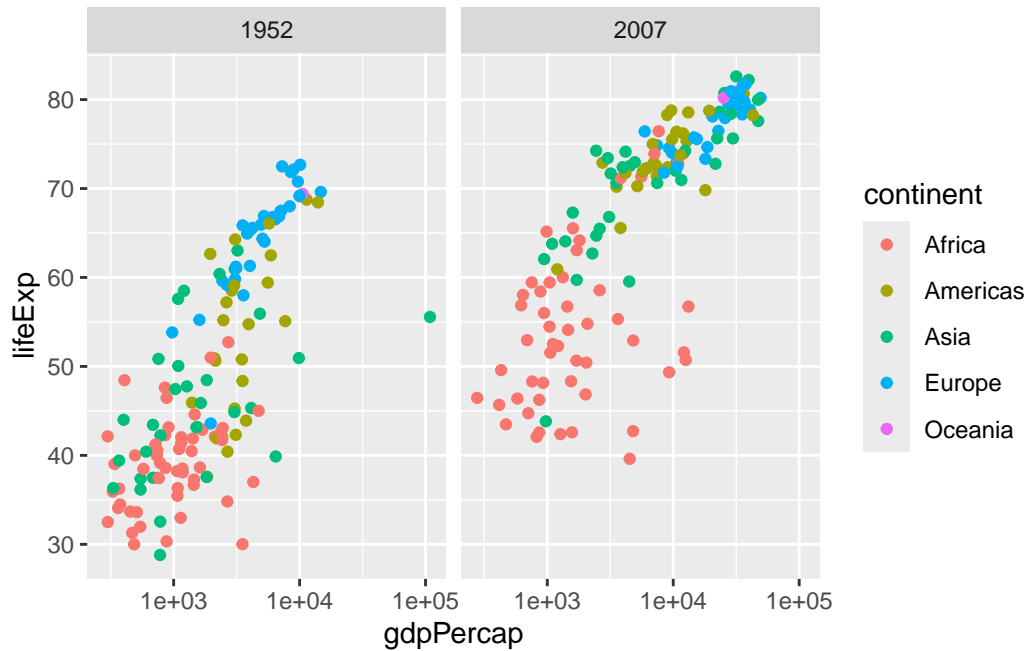
```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
```

```
geom_point() +
facet_wrap(~year) +
coord_cartesian(ylim = c(40, 83))
```



É possível aplicar a escala logarítmica aos eixos também. Isso é útil quando os dados têm uma ampla variação de valores e estamos interessados em destacar diferenças em uma ampla gama de valores, como em dados de renda ou PIB. Para fazer isso, basta utilizar a função `scale*_log10()`:

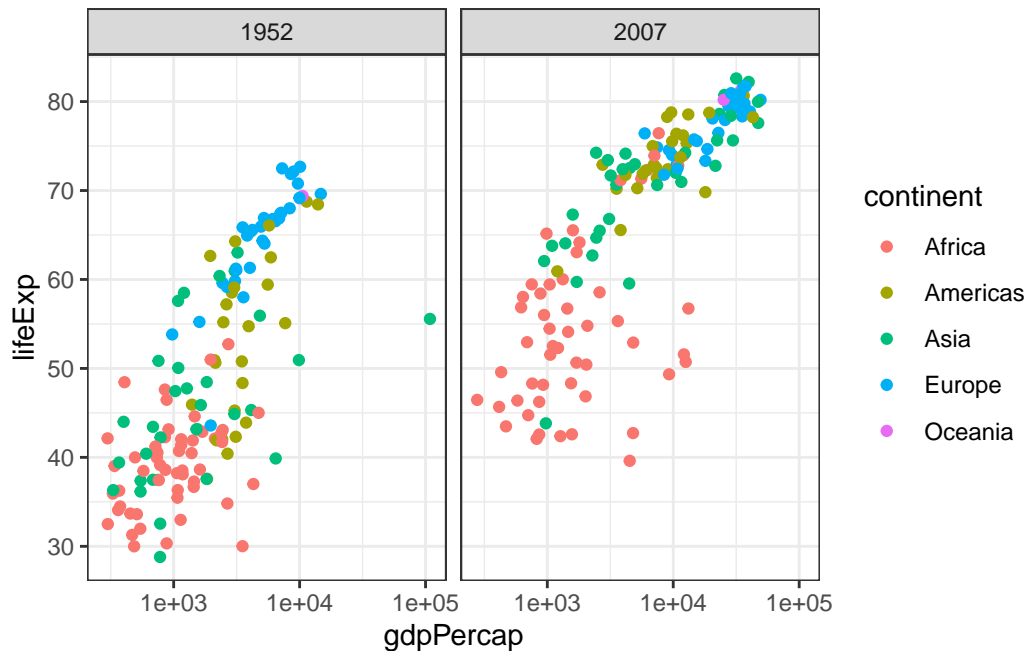
```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
geom_point() +
facet_wrap(~year) +
scale_x_log10()
```



4.2.6 Temas

Os temas controlam os aspectos visuais dos gráficos, como títulos, legendas, eixos e cores de fundo. Os temas pré-definidos, como `theme_bw()`, `theme_minimal()`, `theme_classic()` etc., oferecem estilos visuais consistentes que podem ser aplicados aos gráficos para obter uma aparência específica.

```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  facet_wrap(~year) +
  scale_x_log10() +
  theme_bw()
```

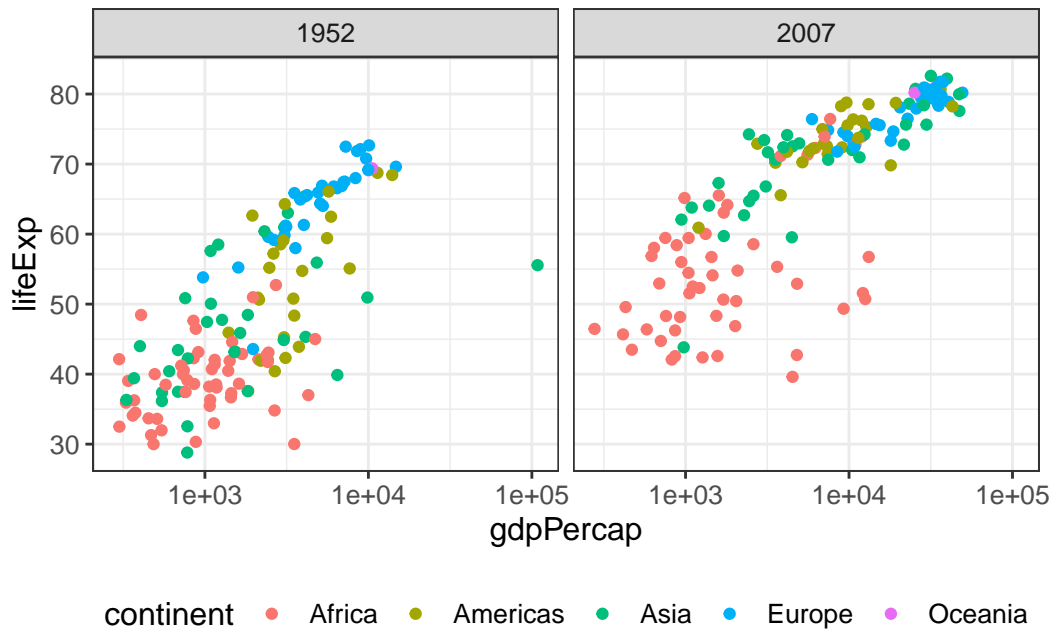


💡 Tente também

Apague a última linha do exemplo acima, digite `theme_`, aperte a tecla tab e experimente os diferentes temas pré-definidos no `ggplot`.

Além disso, podemos definir praticamente todos os aspectos do gráfico. Por exemplo, para trocar a posição da legenda para baixo, podemos usar a função `theme(legend.position = "bottom")`. Para alterar o tamanho da fonte, podemos usar a função `theme(text = element_text(size = 12))`. Essas definições podem ser combinadas em uma única chamada da função `theme()`, veja abaixo.

```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  facet_wrap(~year) +
  scale_x_log10() +
  theme_bw() +
  theme(legend.position = "bottom",
        text = element_text(size = 12))
```

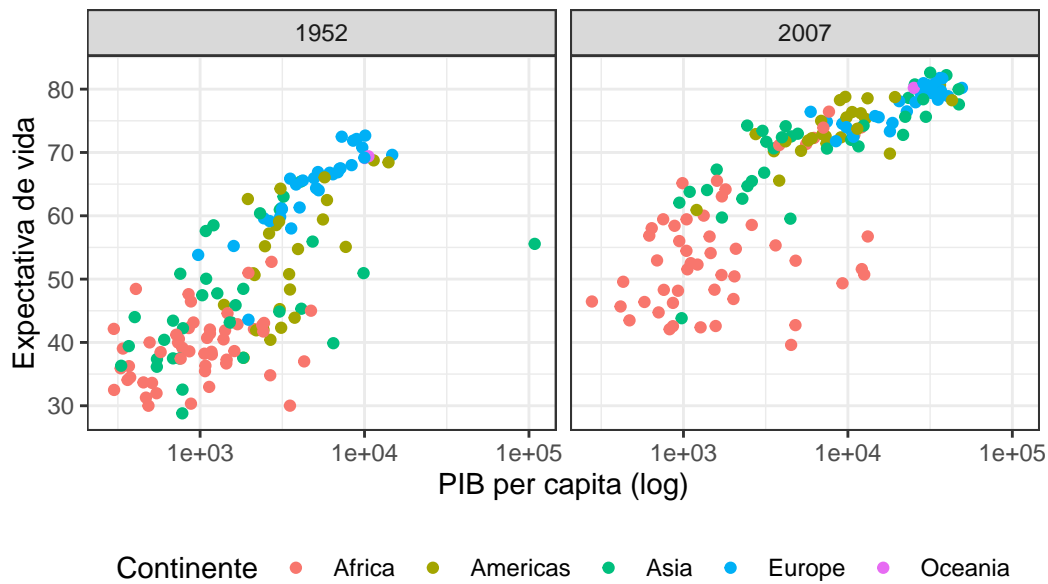


4.2.7 Personalização e Estilização de Gráficos

A função `labs()` é responsável por personalizar rótulos e títulos em gráficos. No exemplo abaixo, renomeamos os eixos x e y e atribuímos um nome mais descritivo para a legenda de cores, que neste caso representa o continente.

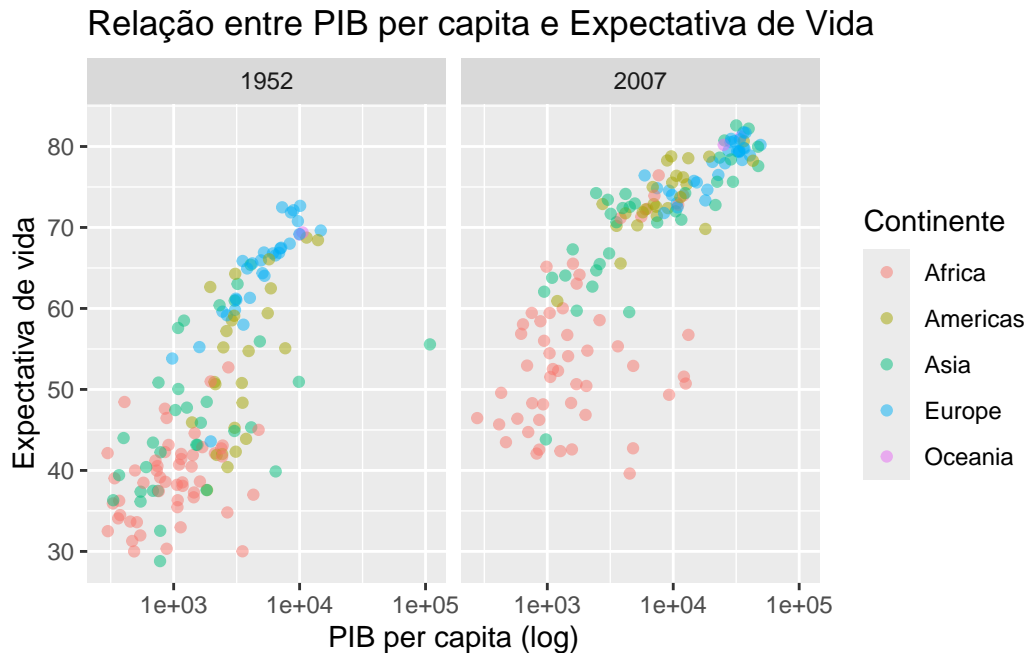
```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  facet_wrap(~year) +
  scale_x_log10() +
  labs(x = "PIB per capita (log)",
       y = "Expectativa de vida",
       color = "Continente",
       title = "Relação entre PIB per capita e Expectativa de Vida") +
  theme_bw() +
  theme(legend.position = "bottom")
```

Relação entre PIB per capita e Expectativa de Vida



O parâmetro `alpha` controla a opacidade dos elementos geométricos, variando de 0 a 1. Por exemplo, `geom_point(alpha = 0.5)` torna os pontos semi-transparentes, o que pode ser útil para visualizar sobreposições de dados em um gráfico de dispersão.

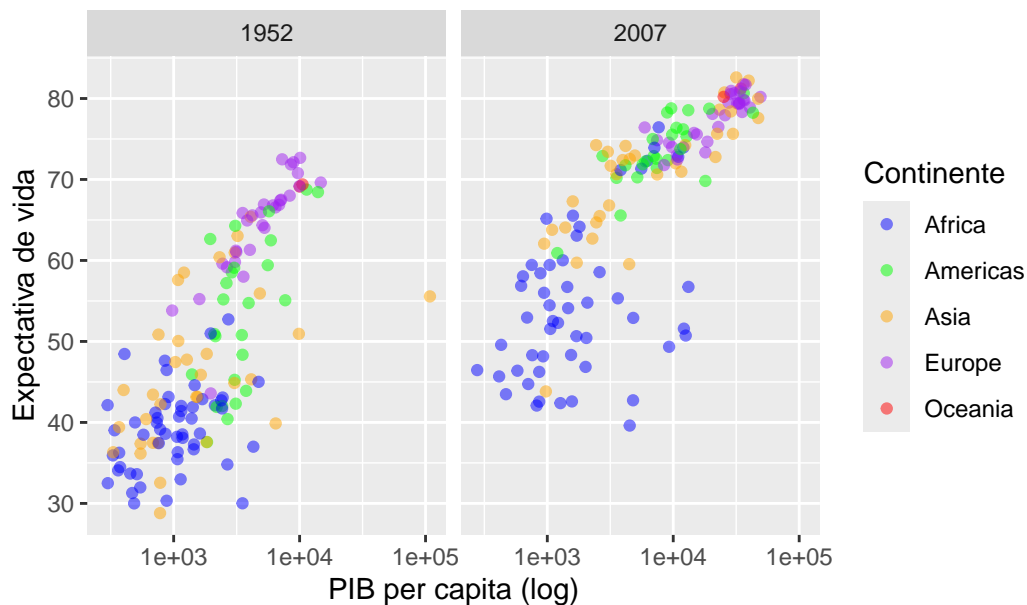
```
ggplot(data = gapminder_anos,
        mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(alpha = 0.5) +
  facet_wrap(~year) +
  scale_x_log10() +
  labs(x = "PIB per capita (log)",
       y = "Expectativa de vida",
       color = "Continente",
       title = "Relação entre PIB per capita e Expectativa de Vida")
```



Para escolher cores específicas para os níveis de uma variável categórica, podemos utilizar a função `scale_color_manual()` para atribuir manualmente cores a cada nível da variável.

```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(alpha = 0.5) +
  facet_wrap(~year) +
  scale_x_log10() +
  scale_color_manual(values = c("blue", "green", "orange", "purple", "red")) +
  labs(x = "PIB per capita (log)",
       y = "Expectativa de vida",
       color = "Continente",
       title = "Relação entre PIB per capita e Expectativa de Vida")
```

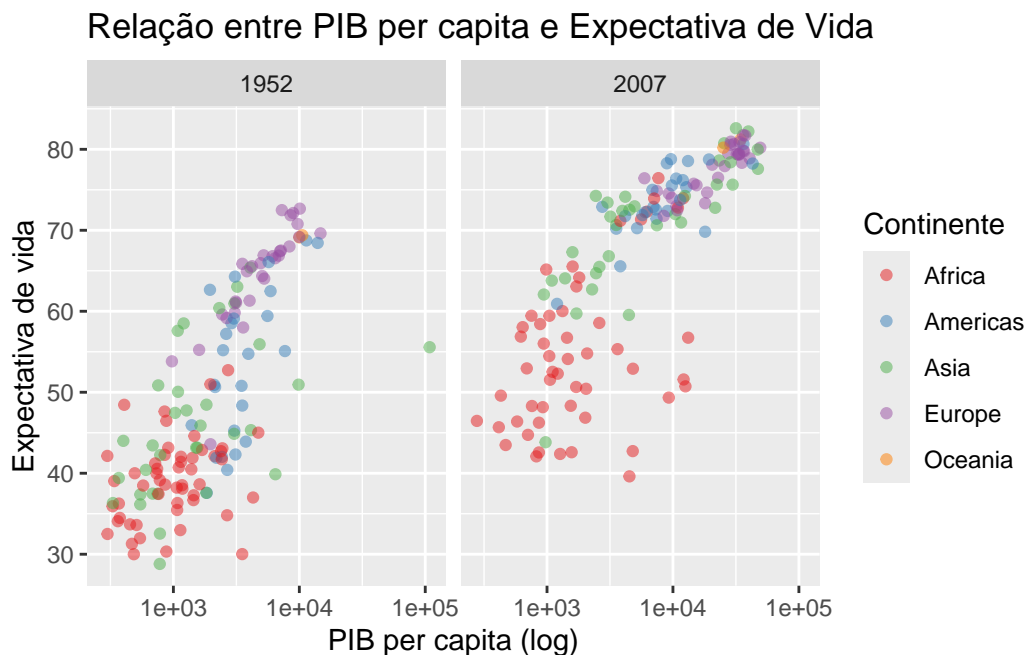
Relação entre PIB per capita e Expectativa de Vida



💡 Outras paletas de cores

Você pode usar paletas de cores do pacote `RColorBrewer` usando a função `scale_color_brewer()`.

```
ggplot(data = gapminder_anos,
       mapping = aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(alpha = 0.5) +
  facet_wrap(~year) +
  scale_x_log10() +
  scale_color_brewer(palette = "Set1")+
  labs(x = "PIB per capita (log)",
       y = "Expectativa de vida",
       color = "Continente",
       title = "Relação entre PIB per capita e Expectativa de Vida")
```

A vantagem de usar as paletas de cores do RColorBrewer é que elas foram cuidadosamente projetadas para serem perceptualmente distintas e adequadas para representar diferentes grupos ou categorias em gráficos. Isso significa que as cores em uma paleta são mais facilmente distinguíveis umas das outras, mesmo quando impressas em preto e branco ou quando vistas por pessoas com deficiências visuais. Veja todas paletas disponíveis [aqui](#)

4.3 Pacotes extras

Existem alguns pacotes extras que funcionam como extensões do `ggplot2`. Apresentamos alguns nessa seção.

4.3.1 O pacote `patchwork`

O pacote `patchwork` é usada para combinar múltiplos gráficos `ggplot2` em uma única visualização. Ele permite criar layouts flexíveis e complexos, adicionando, organizando e ajustando gráficos individualmente.

```
# Instalar o pacote patchwork (apenas se ainda não estiver instalado)
install.packages("patchwork")
```

Depois de carregar o pacote, você pode usar o operador `+` para combinar gráficos `ggplot2` em uma única visualização.

Aqui está um exemplo simples criando dois gráficos separados e, em seguida, são combinados usando o `patchwork`:

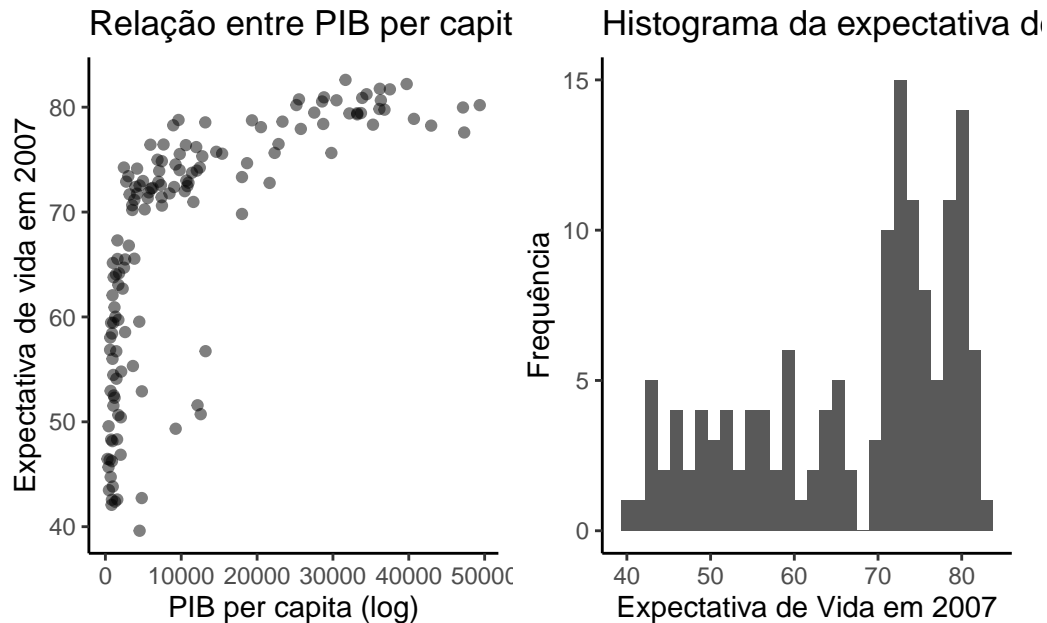
```
library(patchwork)
```

Warning: package 'patchwork' was built under R version 4.2.3

```
plot1 <- ggplot(data = gapminder_2007,
  mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.5) +
  labs(x = "PIB per capita (log)",
    y = "Expectativa de vida em 2007",
    title = "Relação entre PIB per capita e Expectativa de Vida") +
  theme_classic()

plot2 <- ggplot(data = gapminder_2007,
  mapping = aes(lifeExp)) +
  geom_histogram() +
  labs(title = "Histograma da expectativa de vida",
    x = "Expectativa de Vida em 2007",
    y = "Frequência") +
  theme_classic()

plot1 + plot2
```



4.3.2 O pacote ggthemes

O pacote `ggthemes` é uma extensão do `ggplot2` que fornece uma variedade de temas pré-definidos para personalizar a aparência dos gráficos. Veja a [documentação aqui](#).

```
# Instalar o pacote ggthemes (apenas se ainda não estiver instalado)
install.packages("ggthemes")
```

Depois de carregar o pacote, você pode aplicar qualquer um dos temas disponíveis aos seus gráficos `ggplot2` usando a função `theme_*()`. No exemplo abaixo, são criadas três versões do mesmo gráfico com temas diferentes. Os operadores `+` e `/` foram usados para definir como os gráficos serão exibidos.

```
# Carregar o pacote ggthemes
library(ggthemes)
```

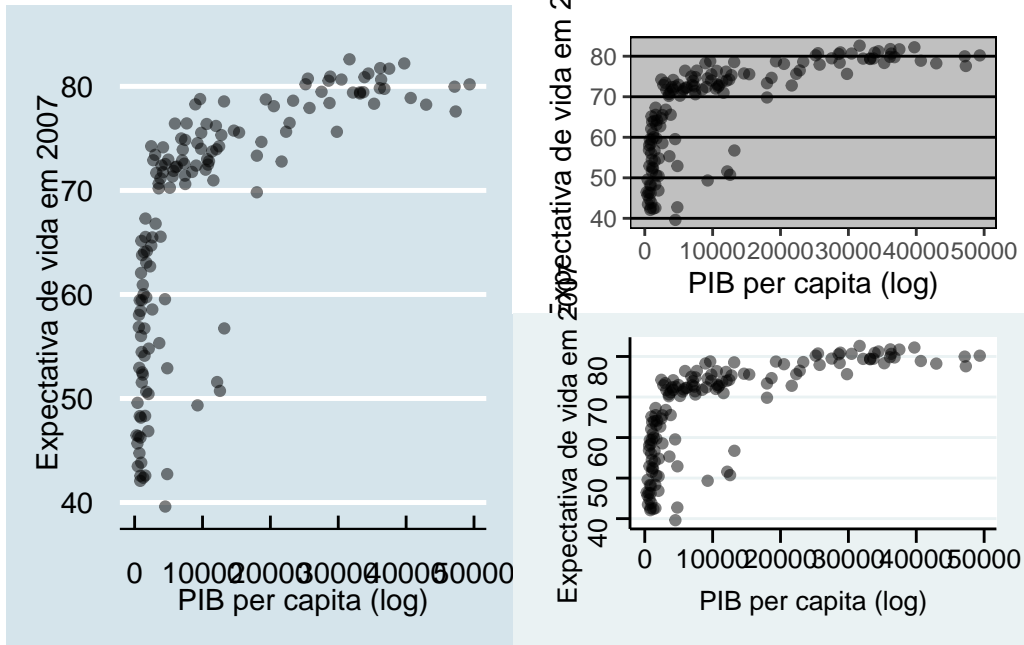
Warning: package 'ggthemes' was built under R version 4.2.3

```
plot0 <- ggplot(data = gapminder_2007,
  mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point(alpha = 0.5) +
```

```
labs(x = "PIB per capita (log)",
     y = "Expectativa de vida em 2007")

plot1 <- plot0 + theme_economist()
plot2 <- plot0 + theme_excel()
plot3 <- plot0 + theme_stata()

plot1 + (plot2 / plot3)
```



4.3.3 O pacote plotly

O pacote `plotly` oferece recursos para criar gráficos interativos. Para adicionar interação ao gráfico fornecido, podemos usar a função `ggplotly()` para converter um gráfico criado com `ggplot2` em um gráfico interativo. Utilize o comando `install.packages("plotly")` caso não tenha o pacote instalado.

```
library(plotly)
```

Warning: package 'plotly' was built under R version 4.2.3

```
grafico <- ggplot(data = gapminder_anos,
                 mapping = aes(x = gdpPercap, y = lifeExp,
                              color = continent, text = country)) +
  geom_point(alpha = 0.5) +
  facet_wrap(~year) +
  scale_x_log10() +
  labs(x = "PIB per capita (log)",
       y = "Expectativa de vida",
       color = "Continente",
       title = "Relação entre PIB per capita e Expectativa de Vida")

ggplotly(grafico)
```

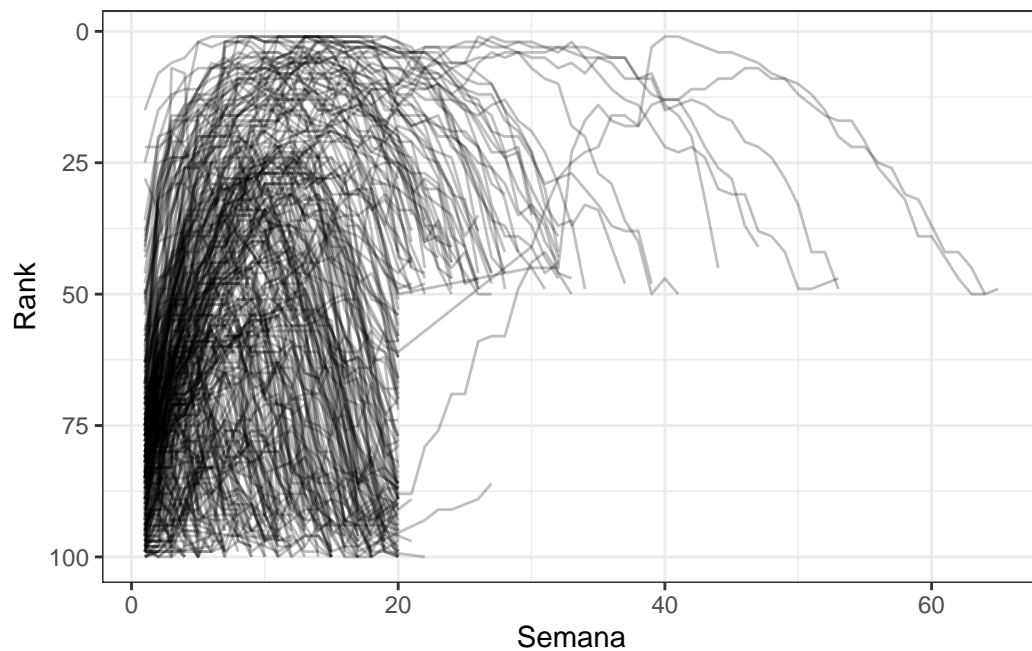
4.4 Dicas extras

Quando você estiver explorando diferentes tipos de gráficos para visualizar seus dados, o site [Data to Viz](#) pode ser uma ferramenta valiosa. Ele fornece uma galeria completa de tipos de gráficos e oferece orientações sobre quando e como usar cada um deles. Além disso, o site oferece exemplos específicos de como criar esses gráficos usando diferentes bibliotecas, como ggplot2 em R e matplotlib em Python.

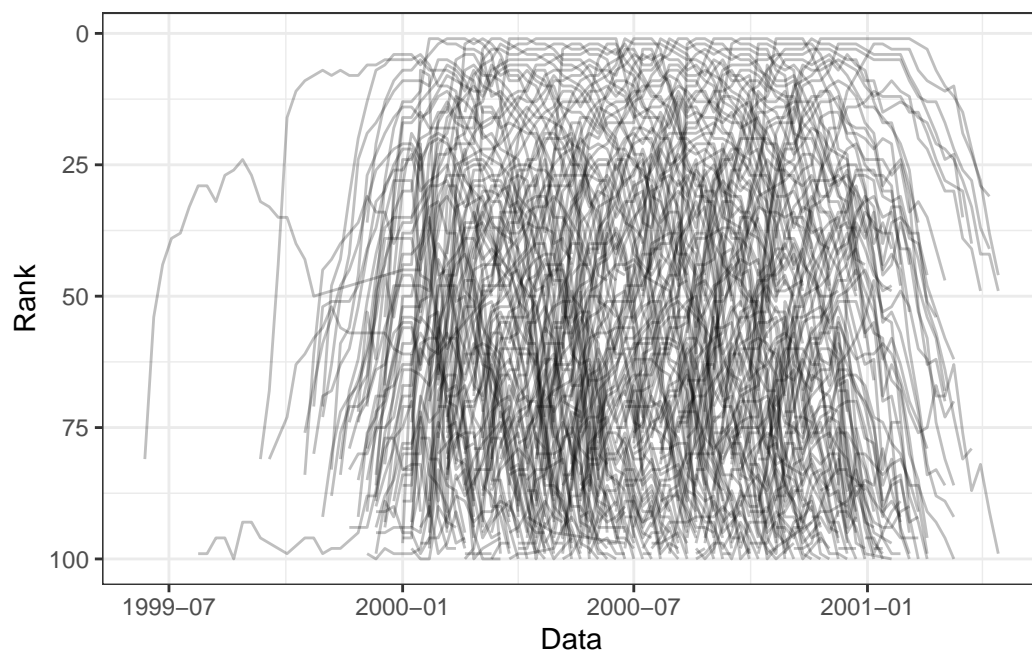
4.5 Exercícios

1. Vamos utilizar os dados da Billboard que foram apresentados na Seção 3.7. Sua tarefa é reproduzir os gráficos abaixo utilizando técnicas de processamento de dados com `dplyr` e visualização de dados com `ggplot`.

a) O gráfico abaixo mostra o histórico de cada musica no ranking ao longo das semanas.

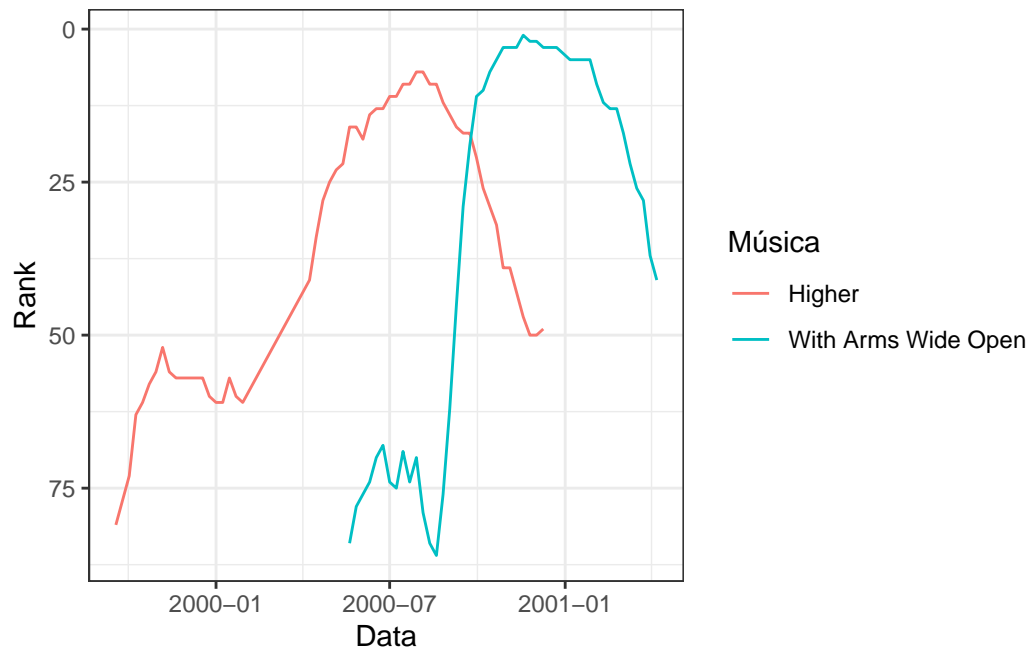


b) O gráfico abaixo é uma pequena alteração do apresentado no item a), no eixo x é apresentada a data que a música entrou no ranking.

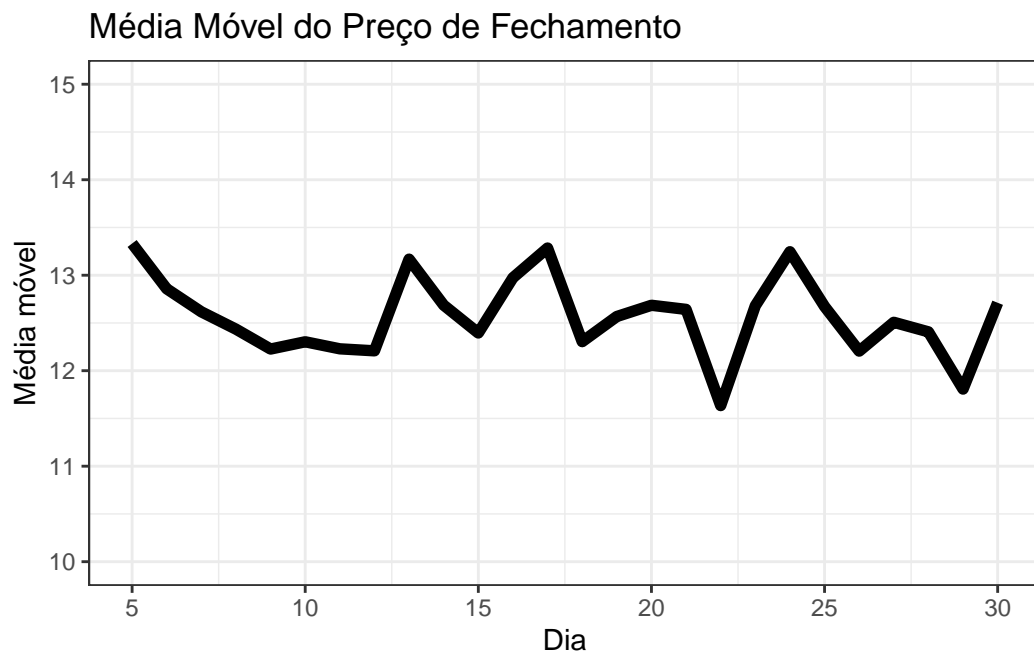


c) O gráfico abaixo mostra o histórico de duas músicas no ranking: “Higher” e “With Arms

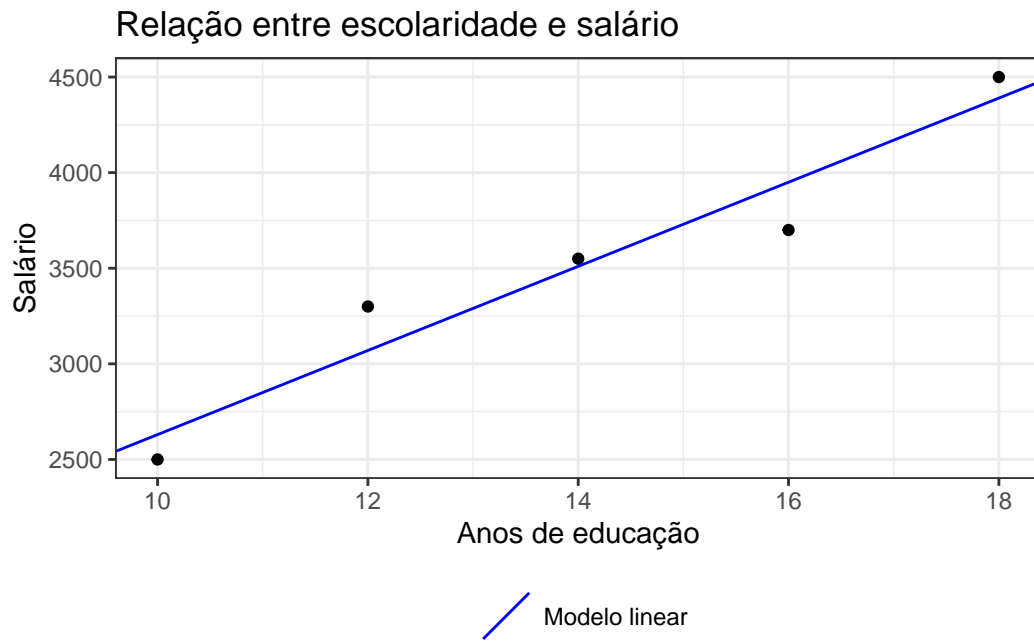
Wide Open”.



2. Utilize os dados gerados na Seção 2.2.1 que representam o preço de fechamento de uma ação e reproduza o gráfico abaixo, mostrando a evolução da média móvel ao longo do tempo.



3. (Desafio) Reproduza o gráfico de dispersão abaixo que foi apresentado na Seção 2.3.



4. Reproduza a figura apresentada na Seção 3.5.6.

Parte II

Aprendendo Python

5 Fundamentos de Python

! Atenção!!!

A partir de agora, todo o código apresentado neste livro está na linguagem Python.

A fonte para a contrução deste material é McKinney (2022).

5.1 Instalação

Siga os passos abaixo para realizar a instalação do Anaconda e do JupyterLab:

1. Baixe e instale o Anaconda a partir do [site oficial](#). Siga as instruções de instalação para o seu sistema operacional específico.
2. Após instalar o Anaconda, abra o Anaconda Navigator e crie um novo ambiente virtual. Navegue até a seção “Environments” e clique em “Create” para adicionar um novo ambiente. Dê um nome ao ambiente e escolha a versão do Python que deseja usar.
3. Após criar o ambiente virtual, ative-o clicando no ambiente virtual recém-criado na lista de ambientes e selecionando “Open Terminal”. No terminal, digite o comando `conda activate nome_do_seu_ambiente` (substitua `nome_do_seu_ambiente` pelo nome do ambiente que você criou).
4. Com o ambiente virtual ativado, instale o JupyterLab digitando `conda install jupyterlab` no terminal.
5. Depois de instalar o JupyterLab, execute-o digitando `jupyter lab` no terminal ou através da interface do Anaconda. Isso abrirá o JupyterLab no seu navegador padrão.
6. No JupyterLab, você pode criar um novo notebook Python clicando no ícone “+” na barra lateral esquerda e selecionando “Python 3” sob o cabeçalho “Notebook”.

5.2 Tipos de dados fundamentais

Em Python, os tipos de dados fundamentais incluem `integer`, `float`, `string` e `boolean`:

- Integers são números inteiros, como 1, 2, -3, etc. - Floats são números decimais, como 3.14, -0.5, etc.
- Strings são sequências de caracteres, como “hello”, “world”, “python”, etc.
- Booleans são valores lógicos que representam verdadeiro (`True`) ou falso (`False`).

Esses tipos de dados são os blocos básicos para representar diferentes tipos de informações em Python, e são amplamente utilizados em programação para realizar operações e manipulações de dados.

5.2.1 O tipo de dado inteiro

Um tipo de dados inteiro (`integer`) em Python representa números inteiros, ou seja, números sem casas decimais. Por exemplo, 5, -10 e 0 são todos exemplos de números inteiros. No Python, os inteiros são representados pela classe `int`. Nos exemplos abaixo realizamos operações básicas com números inteiros.

```
2 + 2
```

4

Na primeira linha, calculamos a soma de 2 com 2.

```
quantidade = 200  
print(quantidade)
```

200

```
type(quantidade)
```

```
<class 'int'>
```

Acima, atribuímos o valor 200 à variável `quantidade` e a imprimimos usando `print(quantidade)`. Por fim, verificamos o tipo de dado da variável `quantidade` com `type(quantidade)`, que retorna `<class 'int'>`, indicando que é um número inteiro.

5.2.2 O tipo de dado ponto flutuante

Um tipo de dado ponto flutuante (`float`) em Python representa números decimais, ou seja, números que podem ter uma parte fracionária. Por exemplo, 3.14, -0.001, e 2.71828 são todos floats. Em Python, os floats são representados pela classe `float`.

```
1.75 + 2**3
```

9.75

```
taxa_juros = 1.25  
  
print(taxa_juros)
```

1.25

```
type(taxa_juros)
```

<class 'float'>

Na primeira linha do exemplo acima, calculamos a soma de 1.75 com 2 elevado à terceira potência. Em seguida, atribuímos o valor 1.25 à variável `taxa_juros` e a imprimimos usando `print(taxa_juros)`. Por fim, verificamos o tipo de dado da variável `taxa_juros` com `type(taxa_juros)`, que retorna `<class 'float'>`, indicando que é um número do tipo ponto flutuante.

5.2.3 O tipo de dado cadeia de caracteres

```
pais = "Brasil"  
  
print(pais)
```

Brasil

```
type(pais)
```

```
<class 'str'>
```

No código acima, criamos uma variável chamada `pais` e atribuímos a ela o valor “Brasil”, que é uma string. Em seguida, imprimimos o valor da variável `pais` usando `print(pais)`, o que exibe “Brasil” na tela. Por fim, verificamos o tipo de dado da variável `pais` com `type(pais)`, que retorna `<class 'str'>`, indicando que é uma string.

5.2.4 O tipo de dado lógico

O tipo de dados lógico, também conhecido como booleano (`bool`), é usado para representar valores de verdadeiro ou falso. No Python, os valores booleanos são `True` e `False`, que representam verdadeiro e falso, respectivamente.

```
1 == 2
```

False

```
5 % 2 == 0
```

False

```
taxa_juros_aumentando = True  
print(taxa_juros_aumentando)
```

True

```
type(taxa_juros_aumentando)
```

```
<class 'bool'>
```

Na primeira linha do exemplo acima, há uma verificação de igualdade entre 1 e 2, que retorna `False` porque 1 não é igual a 2. Em seguida, temos `5 % 2 == 0`, que também retorna `False` porque o resto da divisão de 5 por 2 não é igual a zero. Por fim, temos a variável `taxa_juros_aumentando` atribuída a `True`, indicando que a taxa de juros está aumentando. Ao imprimir e verificar o tipo dessa variável, obtemos `True` como resultado e o tipo `bool`, indicando que é um valor lógico.

5.2.5 Coerção de tipos

A coerção de dados em Python refere-se à conversão forçada de um tipo de dado para outro.

```
str_num = "1.41"  
type(str_num)
```

```
<class 'str'>
```

```
float(str_num)
```

```
1.41
```

Veja que a variável `str_num` é uma string que representa o número 1.41. Inicialmente, seu tipo é verificado usando a função `type`, que retorna `<class 'str'>`, indicando que é uma **string**. Em seguida, usamos a função `float()` para converter explicitamente `str_num` em um `float`. Após a conversão, o valor de `str_num` é “1.41” e seu tipo é alterado para `<class 'float'>`.

A seguir, outros exemplos de coerção.

Coerção para inteiro:

```
num_float = 3.14  
num_int = int(num_float)  
print(num_int)
```

```
3
```

Coerção para lógico:

```
valor_inteiro = 0  
valor_logico = bool(valor_inteiro)  
print(valor_logico)
```

```
False
```

Coerção para string (str):

```
num_float = 3.14
num_str = str(num_float)
print(num_str)
```

3.14

No código acima, estamos convertendo um valor float em string.

5.3 Objetos básicos

Em Python, há três estruturas de dados básicas: listas, tuplas e dicionários:

- As listas são coleções ordenadas e mutáveis de elementos, permitindo a inclusão de itens de diferentes tipos e a modificação dos valores contidos nelas.
- As tuplas são semelhantes às listas, porém são imutáveis, ou seja, não podem ser alteradas após a sua criação.
- Já os dicionários são coleções não ordenadas de pares chave-valor, onde cada valor é associado a uma chave única, proporcionando acesso eficiente aos dados por meio das chaves.

5.3.1 Listas

Em Python, uma lista é uma estrutura de dados que permite armazenar uma coleção ordenada de elementos. Para criar uma lista, utilizamos colchetes [], e os elementos são separados por vírgulas. Podemos instanciar uma lista vazia simplesmente utilizando [] ou a função `list()`. Por exemplo:

```
lista_vazia = [] # Lista vazia
lista_vazia = list() # alternativa

# lista com PIB de países
pib_países = [1800, 2500, 3200, 5600, 6700]
```

No exemplo acima, também temos a lista `pib_países` que armazena o Produto Interno Bruto (PIB) de diferentes países. Para acessar elementos de uma lista em Python, podemos utilizar o índice do elemento desejado dentro de colchetes []. O índice começa do zero para o primeiro elemento, um para o segundo, e assim por diante. Por exemplo:

```
print(pib_países[1])
```

2500

Também podemos acessar os elementos a partir do final da lista utilizando índices negativos, onde -1 representa o último elemento, -2 o penúltimo, e assim por diante:

```
# Acessando o último elemento
ultimo_elemento = pib_países[-1]
print(ultimo_elemento)
```

6700

```
# Acessando o penúltimo elemento
penultimo_elemento = pib_países[-2]
print(penultimo_elemento)
```

5600

Podemos usar o método `append()` se desejamos adicionar um elemento ao final da lista, ou o método `insert()` se queremos adicionar um elemento em uma posição específica. Veja os exemplos de como usar ambos os métodos:

```
# Adicionando elementos ao final da lista usando append()
pib_países.append(2000) # Adiciona o valor 2000 ao final da lista

# Adicionando um elemento em uma posição específica usando insert()
pib_países.insert(1, 1500) # Adiciona o valor 1500 na posição 1 da lista
```

Para verificar o tamanho de uma lista em Python, podemos usar a função `len()`.

```
tamanho_lista = len(pib_países)
print("Tamanho da lista:", tamanho_lista)
```

Tamanho da lista: 7

Para ordenar uma lista, podemos usar o método `sort()` para ordenação *in-place* (ou seja, a lista é modificada) ou a função `sorted()` para retornar uma nova lista ordenada sem modificar a original:


```
# Usando a função sorted() para retornar uma nova lista ordenada
lista_ordenada = sorted(pib_países)
print("Nova lista ordenada:", lista_ordenada)
```

Nova lista ordenada: [1500, 1800, 2000, 2500, 3200, 5600, 6700]

```
# Ordenando a lista usando o método sort()
pib_países.sort()
print("Lista ordenada:", pib_países)
```

Lista ordenada: [1500, 1800, 2000, 2500, 3200, 5600, 6700]

O método `.pop()` é usado para remover e retornar o último elemento de uma lista. Também podemos especificar um índice para remover e retornar um elemento em uma posição específica da lista. Aqui está como usar o método `.pop()`:

```
# Removendo e retornando o último PIB da lista
ultimo_pib = pib_países.pop()
print("Último PIB removido:", ultimo_pib)
```

Último PIB removido: 6700

```
print("Lista atualizada:", pib_países)
```

Lista atualizada: [1500, 1800, 2000, 2500, 3200, 5600]

```
# Removendo e retornando o PIB de um país específico da lista
pib_removido = pib_países.pop(1)
print("PIB removido:", pib_removido)
```

PIB removido: 1800

```
print("Lista atualizada:", pib_países)
```

Lista atualizada: [1500, 2000, 2500, 3200, 5600]

5.3.2 Tuplas

As tuplas são estruturas de dados semelhantes às listas, mas com uma diferença fundamental: elas são imutáveis, ou seja, uma vez criadas, não podem ser modificadas. Elas são representadas por parênteses () em vez de colchetes [].

```
x = (1, 2, 3) # tupla (lista imutável)
print(x)
```

(1, 2, 3)

```
# x[0] = 5 # gera erro!
```

Podemos usar tuplas para representar informações que não devem ser alteradas, como por exemplo, as taxas de câmbio entre moedas. Veja:

```
taxas_cambio = (("USD", "EUR", 0.82), ("USD", "JPY", 105.42), ("EUR", "JPY", 128.64))
```

Neste exemplo, temos uma tupla de tuplas que representam as taxas de câmbio entre o dólar (USD), o euro (EUR) e o iene japonês (JPY) para uma data fixa fictícia. Cada tupla interna contém três elementos: a moeda de origem, a moeda de destino e a taxa de câmbio. Como essas informações não devem ser alteradas, uma tupla é uma escolha apropriada.

Dica

Para acessar os elementos de uma tupla, você pode usar a mesma sintaxe que usa para acessar os elementos de uma lista, ou seja, usando colchetes [] e o índice do elemento desejado. Lembre-se de que os índices em Python começam em 0!

5.3.3 Dicionários

Dicionários em Python são estruturas de dados que permitem armazenar pares de chave-valor. Cada valor é associado a uma chave específica, permitindo o acesso rápido aos dados por meio das chaves, em vez de índices numéricos, como em listas e tuplas. Essa estrutura é útil quando você precisa associar informações de maneira semelhante a um banco de dados, onde você pode buscar informações com base em uma chave específica.

No exemplo abaixo, temos cotações de ações de algumas empresas brasileiras listadas na bolsa de valores.

```
cotacoes_acoes_brasileiras = {
    "PETR4": 36.75,
    "VALE3": 62.40,
    "ITUB4": 34.15,
    "BBDC4": 13.82
}
```

Cada chave é o código de negociação da ação na bolsa, e o valor associado é o preço da ação em uma data fixada. Por exemplo, `cotacoes_acoes_brasileiras["PETR4"]` retornaria o preço da ação da Petrobras.

Alternativamente, você pode criar um dicionário usando a função `dict`:

```
cotacoes_acoes_brasileiras = dict(PETR4=36.12, VALE3=62.40, ITUB4=34.15, BBDC4=13.82)
```

Você pode adicionar novos pares chave-valor a um dicionário ou atualizar os valores existentes. Por exemplo:

```
cotacoes_acoes_brasileiras["ABEV3"] = 12.80
cotacoes_acoes_brasileiras["PETR4"] = 36.75
```

Você pode remover pares chave-valor de um dicionário usando o comando método `pop()`. Por exemplo:

```
valor_removido = cotacoes_acoes_brasileiras.pop("BBDC4")
```

Outros métodos úteis para trabalhar com dicionários são `keys()`, `values()` e `items()` que retornam listas com as chaves, valores e itens do dicionário, respectivamente.

```
cotacoes_acoes_brasileiras.keys() # retorna uma lista contendo todas as chaves
```

```
dict_keys(['PETR4', 'VALE3', 'ITUB4', 'ABEV3'])
```

```
cotacoes_acoes_brasileiras.values() # retorna uma lista contendo todos os valores
```

```
dict_values([36.75, 62.4, 34.15, 12.8])
```

```
cotacoes_acoes_brasileiras.items() # retorna uma lista de tuplas
```

```
dict_items([('PETR4', 36.75), ('VALE3', 62.4), ('ITUB4', 34.15), ('ABEV3', 12.8)])
```

5.4 Fatias (*slices*)

O conceito de fatias, também conhecido como “*slicing*” em inglês, refere-se à técnica de extrair partes específicas de uma sequência, como uma lista, tupla ou string, usando índices. Ao utilizar fatias, você pode selecionar um intervalo de elementos dentro da sequência.

A sintaxe básica para fatias é `sequencia[inicio:fim:passo]`, onde:

- **inicio**: o índice inicial do intervalo a ser incluído na fatia (incluído).
- **fim**: o índice final do intervalo a ser incluído na fatia (excluído).
- **passo**: o tamanho do passo entre os elementos selecionados (opcional).

Considere a lista abaixo.

```
nomes_paises = ["Indonésia", "Índia", "Brasil", "África do Sul", "Alemanha"]
```

Para acessar os três primeiros países, podemos fazer:

```
nomes_paises[:3]
```

```
['Indonésia', 'Índia', 'Brasil']
```

Isso retorna os elementos da lista do índice 0 (inclusivo) ao índice 3 (exclusivo).

Se quisermos acessar os países do segundo ao terceiro:

```
nomes_paises[1:4]
```

```
['Índia', 'Brasil', 'África do Sul']
```

Podemos até mesmo fazer fatias reversas, onde o índice inicial é maior que o índice final, indicando que queremos percorrer a lista de trás para frente. Por exemplo, para acessar os últimos três países:

```
nomes_paises[-3:]
```

```
['Brasil', 'África do Sul', 'Alemanha']
```

Suponha que queremos acessar todos os países, mas pulando de dois em dois:

```
nomes_paises[::2]
```

```
['Indonésia', 'Brasil', 'Alemanha']
```

Neste exemplo, o `::2` indica que queremos começar do início da lista e ir até o final, pulando de dois em dois elementos.

5.5 Condicionais

O `if` e o `else` são estruturas de controle de fluxo em Python, usadas para tomar decisões com base em condições.

O bloco de código dentro do `if` é executado se a condição for avaliada como verdadeira (`True`). Por exemplo:

```
idade = 18
if idade >= 18:
    print("Você é maior de idade.")
```

Você é maior de idade.

O bloco de código dentro do `else` é executado se a condição do `if` for avaliada como falsa (`False`). Por exemplo:

```
idade = 16
if idade >= 18:
    print("Você é maior de idade.")
else:
    print("Você é menor de idade.")
```

Você é menor de idade.

Indentação

Em Python, a indentação é fundamental para definir blocos de código. No exemplo acima, observe que o código dentro do `if` e do `else` está indentado com quatro espaços. Isso indica que essas linhas pertencem ao bloco de código condicional. Se não houver indentação correta, o Python gerará um erro de sintaxe.

5.6 Estruturas repetitivas

As estruturas de repetição são utilizadas para executar um bloco de código repetidamente com base em uma condição específica. Existem duas principais estruturas de repetição em Python: `for` e `while`.

5.6.1 `for`

O loop `for` é utilizado para iterar sobre uma sequência (como uma lista, tupla, dicionário, etc.) e executar um bloco de código para cada item da sequência. Por exemplo:

```
for x in range(0, 20, 3): # lembre da notação dos slices
    print(x)
```

```
0
3
6
9
12
15
18
```

```
for pais in nomes_paises:
    print("País:", pais)
```

```
País: Indonésia
País: Índia
País: Brasil
País: África do Sul
País: Alemanha
```

No exemplo abaixo, temos uma lista de empresas e uma lista de lucros. Usando a função `zip()`, iteramos sobre essas duas listas em paralelo, imprimindo o nome da empresa e seu lucro correspondente. A função `zip()` combina elementos de duas ou mais sequências (como listas, tuplas, etc.) em pares ordenados.

```
empresas = ["Empresa A", "Empresa B", "Empresa C"]
lucros = [100000, 150000, 80000]
```

```
for empresa, lucro in zip(empresas, lucros):  
    print("O lucro da empresa ", empresa, "foi R$", lucro)
```

```
O lucro da empresa  Empresa A foi R$ 100000  
O lucro da empresa  Empresa B foi R$ 150000  
O lucro da empresa  Empresa C foi R$ 80000
```

5.6.2 while

O `while` é uma estrutura de controle de fluxo que executa um bloco de código repetidamente enquanto uma condição especificada for verdadeira.

```
anos = 1  
investimento = 1000  
taxa_de_retorno = 0.05  
  
while anos <= 10:  
    investimento *= (1 + taxa_de_retorno)  
    print("Após", anos, "anos, o investimento vale R$", round(investimento, 2))  
    anos += 1
```

```
Após 1 anos, o investimento vale R$ 1050.0  
Após 2 anos, o investimento vale R$ 1102.5  
Após 3 anos, o investimento vale R$ 1157.62  
Após 4 anos, o investimento vale R$ 1215.51  
Após 5 anos, o investimento vale R$ 1276.28  
Após 6 anos, o investimento vale R$ 1340.1  
Após 7 anos, o investimento vale R$ 1407.1  
Após 8 anos, o investimento vale R$ 1477.46  
Após 9 anos, o investimento vale R$ 1551.33  
Após 10 anos, o investimento vale R$ 1628.89
```

Neste exemplo, o loop calcula o valor do investimento ao longo de 10 anos, considerando um retorno anual de 5%. A cada iteração, o valor do investimento é atualizado multiplicando-se pelo fator de crescimento `(1 + taxa_de_retorno)`.

5.7 Comprehensions

As comprehensions são uma maneira concisa e poderosa de criar coleções em Python, como listas, dicionários e conjuntos, a partir de iteráveis existentes, como listas, dicionários, conjuntos ou sequências. Elas permitem criar essas coleções de forma mais eficiente e legível em comparação com a abordagem tradicional de usar loops. As *comprehensions* podem incluir expressões condicionais para filtrar elementos ou expressões para transformar os elementos durante a criação da coleção.

Por exemplo, você pode criar uma lista de quadrados dos números de 1 a 10 usando uma compreensão de lista:

```
quadrados = [x ** 2 for x in range(1, 11)]
quadrados
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Isso é equivalente a:

```
quadrados = []
for x in range(1, 11):
    quadrados.append(x ** 2)
```

As *comprehensions* podem ser aplicadas a listas, dicionários e conjuntos, e você pode adicionar cláusulas condicionais para filtrar elementos com base em uma condição específica.

Abaixo, as variáveis `linhas` e `colunas` são definidas como intervalos de números de 1 a 3 e de 1 a 2, respectivamente. Em seguida, é utilizada uma compreensão de lista para gerar todos os pares possíveis, combinando cada valor de `linha` com cada valor de `coluna`. Por fim, um loop `for` é usado para iterar sobre a lista de pares e imprimir cada par na saída. O resultado será a impressão de todos os pares ordenados possíveis, combinando os valores de linha e coluna especificados.

```
linhas = range(1, 4)
colunas = range(1, 3)

pares = [(r, c) for r in linhas for c in colunas]

for x in pares:
    print(x)
```


(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)

Neste exemplo abaixo, a palavra “inconstitucionalissimamente” é analisada para contar quantas vezes cada letra aparece. Em seguida, é feito um loop sobre o dicionário resultante para imprimir a contagem de ocorrências de cada letra.

```
palavra = "inconstitucionalissimamente"

frequencia_letras = {letra: palavra.count(letra) for letra in palavra}

for letra, ocorrencias in frequencia_letras.items():
    print("A letra", letra, "ocorre", ocorrencias, "vezes" if ocorrencias > 1 else "vez")
```

```
A letra i ocorre 5 vezes
A letra n ocorre 4 vezes
A letra c ocorre 2 vezes
A letra o ocorre 2 vezes
A letra s ocorre 3 vezes
A letra t ocorre 3 vezes
A letra u ocorre 1 vez
A letra a ocorre 2 vezes
A letra l ocorre 1 vez
A letra m ocorre 2 vezes
A letra e ocorre 2 vezes
```

5.8 Funções

As funções são blocos de código reutilizáveis que realizam uma tarefa específica. Elas aceitam entradas, chamadas de argumentos, e podem retornar resultados.

Em Python, a sintaxe básica de uma função é a seguinte:

```
def nome_da_funcao(argumento1, argumento2, ...):
    # Corpo da função
    # Faça alguma coisa com os argumentos
    resultado = argumento1 + argumento2
```

```
    return resultado
```

Por exemplo, vamos criar uma função em Python chamada `calcular_juros` que calcula o montante final de um investimento com base no valor inicial, na taxa de juros e no número de anos:

```
def calcular_juros(valor_inicial, taxa_juros, anos):  
    montante_final = valor_inicial * (1 + taxa_juros) ** anos  
    return montante_final
```

Agora, definimos valores e chamamos a função:

```
investimento_inicial = 1000 # Valor inicial do investimento  
taxa_juros_anual = 0.05    # Taxa de juros anual (5%)  
anos = 5                  # Número de anos  
resultado = calcular_juros(investimento_inicial, taxa_juros_anual, anos)  
print("O montante final após", anos, "anos será de: R$", round(resultado, 2))
```

O montante final após 5 anos será de: R\$ 1276.28

5.8.1 Função lambda

Uma função lambda em Python é uma função anônima, o que significa que é uma função sem nome. Ela é definida usando a palavra-chave `lambda` e pode ter qualquer número de argumentos, mas apenas uma expressão. A sintaxe básica é a seguinte:

```
lambda argumento1, argumento2, ...: expressao
```

Veja um exemplo de uma função lambda que calcula o quadrado de um número:

```
quadrado = lambda x: x ** 2
```

Neste exemplo, `lambda x: x ** 2` cria uma função que aceita um argumento `x` e retorna `x` ao quadrado. Você pode então usar essa função da mesma forma que qualquer outra função. Por exemplo:

```
resultado = quadrado(5)  
print(resultado)
```

As funções lambda são frequentemente usadas em situações em que você precisa de uma função temporária e simples, como em operações de mapeamento, filtragem e ordenação de dados.

5.9 Classes e objetos

Uma classe é uma estrutura que define o comportamento e as propriedades de um tipo de objeto. Podemos pensar em uma classe como uma representação de um conceito abstrato, como uma transação financeira ou um tipo específico de investimento.

Por exemplo, podemos criar uma classe chamada `Transacao` para representar uma transação financeira, com propriedades como o valor da transação, a data e o tipo de transação.

Um objeto, por outro lado, é uma instância específica de uma classe. Ele representa uma entidade concreta com suas próprias características e comportamentos. Continuando com o exemplo da classe `Transacao`, podemos criar objetos individuais para representar transações específicas, como a compra de ações de uma empresa em uma determinada data.

```
class Transacao:
    def __init__(self, valor, data, tipo):
        self.valor = valor
        self.data = data
        self.tipo = tipo

    def print_info(self):
        print(f"Tipo da transação: {self.tipo}, Valor: R${self.valor}, Data:{self.data}")
```

Neste exemplo, a classe `Transacao` possui um método especial `__init__` que é chamado quando um novo objeto é criado. Esse método inicializa as propriedades do objeto com os valores fornecidos como argumentos. Além disso, a classe possui um método chamado `print_info`, que imprime as informações da transação, incluindo o tipo, o valor e a data. Esse método também recebe `self` como parâmetro para acessar os atributos da instância atual da classe. Ao chamar `print_info()` em um objeto `Transacao`, ele exibirá as informações formatadas da transação.

A seguir, criamos dois objetos da classe `Transacao`, `transacao1` e `transacao2`, e acessamos suas propriedades para obter informações sobre as transações.

```
# Criando objetos da classe Transacao
transacao1 = Transacao(valor=1000, data="2024-03-11", tipo="Compra de ações")
transacao2 = Transacao(valor=500, data="2024-03-12", tipo="Venda de ações")

# Acessando as propriedades dos objetos
```

```
print("Valor da transação 1:", transacao1.valor)
```

Valor da transação 1: 1000

```
print("Data da transação 2:", transacao2.data)
```

Data da transação 2: 2024-03-12

```
# Acessando métodos dos objetos  
transacao1.print_info()
```

Tipo da transação: Compra de ações, Valor: R\$1000, Data:2024-03-11

```
transacao2.print_info()
```

Tipo da transação: Venda de ações, Valor: R\$500, Data:2024-03-12

5.10 Exercícios

1. Trabalhando com tipos de dados básicos

a) Crie variáveis para representar dados econômicos, como o PIB de pelo menos três países, taxas de inflação ou taxas de desemprego. Use valores recentes e históricos para criar um conjunto diversificado de dados econômicos que representem diferentes contextos econômicos ao redor do mundo.

b) Realize operações matemáticas básicas com esses dados, como calcular médias, taxas de crescimento ou proporções. Por exemplo, você pode calcular médias dos valores do PIB, taxas de crescimento do PIB ao longo do tempo, proporções entre diferentes indicadores econômicos (como o PIB per capita em relação ao PIB total), entre outras operações.

2. Durante a análise de dados, pode ser necessário converter entre diferentes tipos de dados. Utilize os tipos de dados fundamentais (integers, floats, strings) e aplique coerção de tipos conforme necessário. Considere, por exemplo que as variáveis representando o PIB foram dadas em formato **string**. Então, converta um tipo **float** para computar a média dos PIBs considerados. Depois disso, converta novamente os valores de PIB em uma **string** para formatação de saída ser no seguinte formato 1,111%.

3. Utilize o código abaixo para gerar um pandas dataframe que representa o preço de fechamento de uma ação

```
import pandas as pd

# Criando uma lista de datas
datas = pd.date_range(start='2023-01-01', end='2023-12-31', freq='B') # Frequência 'B' pa

# Criando uma série de preços de fechamento simulados
import numpy as np
np.random.seed(0) # Define a semente aleatória para reprodutibilidade
precos_fechamento = np.random.normal(loc=50, scale=5, size=len(datas)) # Simulando preços

# Criando o DataFrame
df_precos_acoes = pd.DataFrame({'Data': datas, 'Preço de Fechamento': precos_fechamento})

# Exibindo as primeiras linhas do DataFrame
print(df_precos_acoes.head())
```

	Data	Preço de Fechamento
0	2023-01-02	58.820262
1	2023-01-03	52.000786
2	2023-01-04	54.893690
3	2023-01-05	61.204466
4	2023-01-06	59.337790

a) Obtenha os preços de fechamento da ação durante do mês de janeiro determinado.

b) Obtenha os preços de fechamento da ação entre 2023-06-01 e 2023-12-31.

c) Obtenha os preços de fechamento da ação às sextas-feiras ao longo de todo o período. Calcule o preço de fechamento médio às segundas-feiras e compare com o das sextas-feiras. Qual deles apresenta maior desvio padrão?

```
# Criando uma coluna para armazenar o dia da semana
df_precos_acoes['Dia da Semana'] = df_precos_acoes['Data'].dt.weekday

# Exibindo as primeiras linhas do DataFrame com a nova coluna
print(df_precos_acoes.head())
```

	Data	Preço de Fechamento	Dia da Semana
0	2023-01-02	58.820262	0

1	2023-01-03	52.000786	1
2	2023-01-04	54.893690	2
3	2023-01-05	61.204466	3
4	2023-01-06	59.337790	4

4. Considere a lista armazenada na variável `pib_anos` abaixo e utilize uma compreensão de lista para calcular o crescimento percentual do PIB em relação ao ano anterior para cada país.

```
# Lista de PIB de um país nos últimos cinco anos
pib_anos = [1000, 1200, 800, 1500, 2000] # Exemplo de valores fictícios para o PIB
```

5. Escreva uma função chamada `calcular_ipc` que receba três argumentos:

- `cesta_de_produtos`: Um dicionário que mapeia cada produto a sua quantidade na cesta.
- `precos_atual`: Um dicionário que mapeia cada produto ao seu preço atual.
- `precos_base`: Um dicionário que mapeia cada produto ao seu preço base (preço de referência). O IPC é calculado utilizando a seguinte fórmula:

$$IPC = \sum_i \frac{\text{preço atual do produto}_i}{\text{preço base do produto}_i} \times \text{quantidade do produto}_i.$$

A função deve retornar o valor calculado do IPC. Use o código abaixo para testar sua função.

```
# Cesta de produtos com suas respectivas quantidades
cesta_de_produtos = {'arroz': 1, 'feijao': 2, 'carne': 3}
# Preços atuais dos produtos
precos_atual = {'arroz': 5, 'feijao': 8, 'carne': 12}
# Preços base dos produtos
precos_base = {'arroz': 4, 'feijao': 7, 'carne': 10}

# Chamada da função para calcular o IPC
ipc = calcular_ipc(cesta_de_produtos, precos_atual, precos_base)
print("O Índice de Preços ao Consumidor (IPC) é:", ipc)
```

6. Você está encarregado de desenvolver um sistema para registrar e gerenciar transações de compra e venda de ações, além de calcular informações importantes sobre a carteira de investimentos. Para isso, você deve implementar duas classes em Python: `Transacao` e `Carteira`.

A classe `Transacao` representa uma única transação de compra ou venda de ações. Ela possui os seguintes atributos:

- **data:** uma string representando a data da transação no formato 'AAAA-MM-DD'. tipo: uma string indicando o tipo da transação, que pode ser 'compra' ou 'venda'.
- **valor:** um número inteiro representando a quantidade de ações transacionadas.

A classe **Carteira** representa a carteira de investimentos do usuário, que contém várias transações de ações. Ela possui os seguintes atributos: - **transacoes:** uma lista que armazena todas as transações de ações realizadas.

Além disso, a classe **Carteira** possui os seguintes métodos:

- **adicionar_transacao(transacao):** adiciona uma nova transação à carteira.
- **calcular_posicao_atual(valor_atual_acao):** calcula a posição atual da ação na carteira com base no valor atual da ação.
- **calcular_valor_presente(valor_atual_acao):** calcula o valor presente da ação na carteira com base no valor atual da ação.

- Implemente as classes **Transacao** e **Carteira** com os atributos e métodos descritos acima.
- Crie três instâncias da classe **Transacao** para representar diferentes transações de compra e venda de ações.
- Crie uma instância da classe **Carteira** e adicione as transações criadas à carteira.
- Teste os métodos da classe **Carteira**, utilizando os exemplos fornecidos no código de teste abaixo.

```
# Criando algumas transações
transacao1 = Transacao('2024-03-18', 'compra', 10) # Compra de 10 ações
transacao2 = Transacao('2024-03-19', 'compra', 5)  # Compra de mais 5 ações
transacao3 = Transacao('2024-03-20', 'venda', 8)   # Venda de 8 ações

# Criando uma carteira e adicionando as transações
carteira = Carteira()
carteira.adicionar_transacao(transacao1)
carteira.adicionar_transacao(transacao2)
carteira.adicionar_transacao(transacao3)

# Valor atual da ação (hipotético)
valor_atual_acao = 50

# Testando os métodos da classe Carteira
posicao_atual = carteira.calcular_posicao_atual(valor_atual_acao)
valor_presente = carteira.calcular_valor_presente(valor_atual_acao)

# Exibindo os resultados
```

```
print("Posição atual da ação na carteira:", posicao_atual)  
print("Valor presente na carteira:", valor_presente)
```


6 Processamento e visualização de dados

6.1 Instalação de bibliotecas

A instalação de bibliotecas em Python é essencial para expandir a funcionalidade da linguagem. Existem várias maneiras de instalar bibliotecas, mas a mais comum é usando um gerenciador de pacotes. O pip é o gerenciador de pacotes padrão para Python e geralmente acompanha a instalação do Python.

Para instalar uma biblioteca com pip, abra o terminal ou prompt de comando e digite o seguinte comando:

```
pip install nome_da_biblioteca
```

Substitua `nome_da_biblioteca` pelo nome da biblioteca que você deseja instalar.

6.2 Processamento de dados numéricos

O NumPy (Numerical Python) é uma biblioteca essencial para computação numérica em Python. Ele fornece estruturas de dados eficientes para trabalhar com arrays multidimensionais e funções matemáticas poderosas para manipulação de dados.

Para instalar o NumPy, você pode usar o pip, que é o gerenciador de pacotes padrão do Python:

```
pip install numpy
```

O principal objeto em NumPy é o array multidimensional. Você pode criar arrays NumPy usando a função `numpy.array()` e realizar operações matemáticas básicas com eles:

```
import numpy as np

# Criando um array NumPy
arr = np.array([1, 2, 3, 4, 5])
```

```
# Operações matemáticas básicas
print("Soma:", np.sum(arr))
```

Soma: 15

```
print("Média:", np.mean(arr))
```

Média: 3.0

Além das operações básicas, o NumPy oferece funções universais (**ufuncs**) para aplicar operações em todos os elementos de um array de uma vez:

```
# Funções universais (ufuncs)
arr = np.array([1, 2, 3, 4, 5])
print("Quadrado de cada elemento:", np.square(arr))
```

Quadrado de cada elemento: [1 4 9 16 25]

```
print("Exponencial de cada elemento:", np.exp(arr))
```

Exponencial de cada elemento: [2.71828183 7.3890561 20.08553692 54.59815003 148.4131596]

A seguir, apresentamos três exemplos práticos de utilização das funções do NumPy.

Exemplo 1: Cálculo de Estatísticas Descritivas

O NumPy pode ser usado para calcular estatísticas descritivas, como média, mediana, desvio padrão, mínimo e máximo de séries temporais de dados econômicos, como o preço das ações de uma empresa ao longo do tempo.

```
import numpy as np

# Preço das ações de uma empresa ao longo do tempo (em dólares)
precos = np.array([100, 102, 105, 110, 108, 115, 120])

# Calculando estatísticas descritivas
print("Média:", np.mean(precos))
```

Média: 108.57142857142857

```
print("Desvio padrão:", np.std(precos))
```

Desvio padrão: 6.58693821908486

```
print("Máximo:", np.max(precos))
```

Máximo: 120

```
print("Mínimo:", np.min(precos))
```

Mínimo: 100

Exemplo 2: Análise de Séries Temporais O NumPy é útil para manipulação e análise de séries temporais. Por exemplo, você pode usar NumPy para calcular a taxa de retorno de um investimento ao longo do tempo ou para suavizar uma série temporal usando médias móveis.

```
precos = np.array([100, 102, 105, 110, 108, 115, 120])
# Calcular a taxa de retorno de um investimento ao longo do tempo
retornos = np.diff(precos) / precos[:-1] * 100
print("Taxa de retorno:", retornos)
```

Taxa de retorno: [2. 2.94117647 4.76190476 -1.81818182 6.48148148 4.34782609]

Dica

A função `np.diff` em NumPy é usada para calcular a diferença entre elementos consecutivos ao longo de um determinado eixo de um array. *Atenção:* O tamanho do retorno da função `np.diff` será sempre menor que o tamanho do vetor original de entrada por um elemento. Por exemplo, se tivermos um vetor unidimensional com n elementos, a função `np.diff` retornará um vetor com $n - 1$ elementos, pois não há diferença para o último elemento.

```
precos = np.array([100, 102, 105, 110, 108, 115, 120])
# Suavizar uma série temporal usando médias móveis
tamanho_janela = 3
media_movel = np.convolve(precos, np.ones(tamanho_janela) / tamanho_janela, mode='valid')
```

```
print("Médias móveis:", media_movel)
```

Médias móveis: [102.33333333 105.66666667 107.66666667 111. 114.33333333]

Dica

A função `np.convolve` em NumPy é usada para realizar a convolução entre duas sequências, representadas por dois vetores unidimensionais. A convolução é uma operação matemática que combina duas funções para produzir uma terceira função que representa a quantidade de sobreposição entre elas conforme uma delas é deslocada ao longo do eixo. A sintaxe básica da função é `np.convolve(a, b, mode='full')`, onde `a` e `b` são os dois vetores unidimensionais a serem convolvidos e `mode` é um parâmetro opcional que define o modo de convolução. Os modos mais comuns são:

- `'full'`: Retorna a saída completa da convolução. O comprimento do resultado será `len(a) + len(b) - 1`.
- `'valid'`: Retorna apenas pontos onde as sequências se sobrepõem completamente. O comprimento do resultado será `max(len(a), len(b)) - min(len(a), len(b)) + 1`.
- `'same'`: Retorna a saída do mesmo tamanho que o vetor de entrada mais longo. O comprimento do resultado será `max(len(a), len(b))`.

No exemplo anterior, a função `np.convolve` foi usada aqui para calcular a média móvel dos preços. Nesse caso, a primeira sequência é o vetor de preços e a segunda sequência é um vetor de 1s dividido pelo tamanho da janela de média móvel. Isso cria uma sequência que representa uma média ponderada dos valores.

Exemplo 3: Simulação Monte Carlo O NumPy pode ser usado para realizar simulações Monte Carlo, que são amplamente utilizadas na modelagem financeira e na avaliação de risco. Por exemplo, você pode simular o desempenho de uma carteira de investimentos ao longo do tempo sob diferentes cenários de mercado.

```
# Simulação Monte Carlo do desempenho de uma carteira de investimentos
num_simulacoes = 1000
num_anos = 10
retorno_medio = 0.08
volatilidade = 0.15

# Gerar retornos aleatórios usando uma distribuição normal
retornos = np.random.normal(retorno_medio, volatilidade, size=(num_simulacoes, num_anos))
```

```
# Calcular o valor final da carteira para cada simulação
investimento_inicial = 10000
valores_finais = investimento_inicial * np.cumprod(1 + retornos, axis=1)

# Estatísticas descritivas dos valores finais da carteira
print("Valor final médio:", np.mean(valores_finais[:,-1]))
```

Valor final médio: 22277.71467156478

```
print("Desvio padrão dos valores finais:", np.std(valores_finais[:,-1]))
```

Desvio padrão dos valores finais: 10633.069728343136

Mais referências sobre NumPy:

- **Documentação oficial do NumPy:** <https://numpy.org/doc/stable/> A documentação oficial do NumPy contém informações detalhadas sobre todas as funções e métodos disponíveis, além de tutoriais e exemplos.
- **NumPy Quickstart Tutorial:** <https://numpy.org/doc/stable/user/quickstart.html> Este tutorial rápido fornece uma introdução rápida ao NumPy e suas funcionalidades básicas.

6.3 Análise e processamento de dados

O pandas é uma biblioteca de código aberto amplamente utilizada em Python para análise e manipulação de dados. Ele fornece estruturas de dados flexíveis e ferramentas poderosas para trabalhar com dados estruturados, facilitando a análise, limpeza e preparação de dados para diversas aplicações, como ciência de dados, finanças, pesquisa acadêmica e muito mais.

6.4 O que é pandas?

Pandas é uma biblioteca Python de código aberto que oferece estruturas de dados de alto desempenho e ferramentas de análise de dados. O pandas foi projetado para lidar com as complexidades do mundo real em análise de dados, oferecendo uma interface simples e intuitiva para trabalhar com dados tabulares.

Pandas é amplamente utilizado em análise de dados devido à sua capacidade de:

- Importar e exportar dados de uma variedade de fontes, incluindo arquivos CSV, Excel, SQL, JSON, HDF5 e mais.
- Manipular dados de forma eficiente, incluindo indexação, filtragem, agregação e limpeza.
- Realizar operações estatísticas e matemáticas em dados, como média, soma, desvio padrão, correlação, etc.

As duas principais estruturas de dados fornecidas pelo pandas são series e dataframes.

6.4.1 Séries

Uma **Série** é uma estrutura de dados unidimensional que pode conter qualquer tipo de dados, como inteiros, floats, strings, entre outros. Cada elemento em uma Série possui um rótulo único chamado de índice. A Série é semelhante a uma lista ou array unidimensional em Python, mas fornece recursos adicionais, como operações vetorizadas e alinhamento automático de dados com base nos rótulos de índice.

Suponha que temos uma Série representando os preços diários de uma ação:

Data	Preço
2024-03-18	100
2024-03-19	105
2024-03-20	98
2024-03-21	102

Podemos criar uma Serie pandas para representar esses dados:

```
import pandas as pd

# Dados dos preços da ação
data = ['2024-03-18', '2024-03-19', '2024-03-20', '2024-03-21']
precos = [100, 105, 98, 102]

# Criando uma Série pandas
serie_precos_acao = pd.Series(precos, index=pd.to_datetime(data), name='Preço da Ação')
print(serie_precos_acao)
```

```
2024-03-18    100
2024-03-19    105
2024-03-20     98
2024-03-21    102
Name: Preço da Ação, dtype: int64
```

6.4.2 DataFrame

DataFrame é uma estrutura de dados bidimensional semelhante a uma tabela de banco de dados ou uma planilha do Excel. Ele é composto por linhas e colunas, onde cada coluna pode conter um tipo de dado diferente. Cada coluna e linha em um DataFrame possui um rótulo exclusivo chamado de índice e nome, respectivamente. O DataFrame permite realizar uma ampla gama de operações de manipulação e análise de dados, como indexação, filtragem, agregação, limpeza, entre outras.

Suponha que temos um DataFrame representando os preços diários de várias ações. Podemos criar um DataFrame pandas para representar esses dados. Veja no exemplo abaixo.

```
import pandas as pd
import numpy as np

# Dados dos preços das ações
data = ['2024-03-18', '2024-03-19', '2024-03-20', '2024-03-21']
precos_acoes = {
    'Ação 1': [100, 105, np.nan, 102],
    'Ação 2': [50, 52, 48, 49],
    'Ação 3': [75, np.nan, 72, 74]
}

# Criando um DataFrame pandas
df_precos_acoes = pd.DataFrame(precos_acoes, index=pd.to_datetime(data))
print(df_precos_acoes)
```

	Ação 1	Ação 2	Ação 3
2024-03-18	100.0	50	75.0
2024-03-19	105.0	52	NaN
2024-03-20	NaN	48	72.0
2024-03-21	102.0	49	74.0

6.4.3 Principais funcionalidades

A função `df.isna()` é uma função fornecida pelo pandas em um DataFrame (`df`) que retorna uma matriz booleana indicando se cada elemento do DataFrame é um valor ausente (NaN).

Quando aplicada a um DataFrame, a função `isna()` retorna um DataFrame com o mesmo formato, onde cada valor é substituído por `True` se for NaN e `False` caso contrário.

Isso é útil para identificar rapidamente os valores ausentes em um DataFrame e realizar operações de limpeza ou tratamento de dados, como preenchimento de valores ausentes ou remoção de linhas ou colunas contendo esses valores.

Se aplicarmos `df_precos_acoes.isna()`, obteremos:

```
df_precos_acoes.isna()
```

	Ação 1	Ação 2	Ação 3
2024-03-18	False	False	False
2024-03-19	False	False	True
2024-03-20	True	False	False
2024-03-21	False	False	False

Para contar a quantidade de NaN em cada coluna, combine `is.na()` com `sum()`:

```
df_precos_acoes.isna().sum()
```

```
Ação 1    1
Ação 2    0
Ação 3    1
dtype: int64
```

O método `dropna()` no pandas é usado para remover linhas ou colunas que contenham valores ausentes (NaN).

```
df_precos_acoes.dropna()
```

	Ação 1	Ação 2	Ação 3
2024-03-18	100.0	50	75.0
2024-03-21	102.0	49	74.0

O parâmetro `subset` é usado para especificar em quais colunas ou linhas o pandas deve procurar por valores ausentes antes de remover. Quando usamos `df.dropna(subset=["Ação 3"])`, estamos instruindo o pandas a remover todas as linhas onde houver um valor ausente na coluna “Ação 3”.

```
df_precos_acoes.dropna(subset=["Ação 3"])
```


	Ação 1	Ação 2	Ação 3
2024-03-18	100.0	50	75.0
2024-03-20	NaN	48	72.0
2024-03-21	102.0	49	74.0

Na função `dropna()`, o parâmetro `inplace=True` especifica que a modificação deve ser feita diretamente no DataFrame original, em vez de retornar um novo DataFrame sem os valores ausentes. Quando `inplace=True` é usado com `dropna()`, o DataFrame original é modificado e as linhas ou colunas com valores ausentes são removidas permanentemente.

```
df_precos_acoes.dropna(inplace = True)
```

A função `fillna()` no pandas é usada para preencher valores ausentes (NaN) em um DataFrame com um valor específico.

Considere o seguinte DataFrame `df` que representa os dados de clientes de um banco com alguns dados faltantes:

```
import pandas as pd
import numpy as np

dados = {'Nome': ['João', 'Maria', 'Pedro', 'Ana', 'Mariana'],
        'Idade': [25, 30, np.nan, 40, 35],
        'Renda Mensal': [5000, 6000, np.nan, 4500, 5500],
        'Limite de Crédito': [10000, np.nan, 8000, 12000, np.nan]}
df_clientes = pd.DataFrame(dados)
```

Neste exemplo,

- Os valores ausentes na coluna “Idade” foram preenchidos com a mediana das idades existentes no DataFrame.
- Os valores ausentes na coluna “Limite de Crédito” foram preenchidos com a moda dos limites de crédito existentes no DataFrame.
- Os valores ausentes na coluna “Renda Mensal” foram preenchidos com a média das rendas mensais existentes no DataFrame.

```
# Preenchendo valores ausentes na coluna 'Idade' com a mediana das idades
df_clientes['Idade'] = df_clientes['Idade'].fillna(df_clientes['Idade'].median())

# Preenchendo valores ausentes na coluna 'Limite de Crédito' com a moda dos limites de crédito
df_clientes['Limite de Crédito'] = df_clientes['Limite de Crédito'].fillna(df_clientes['Limite de Crédito'].mode()[0])

# Preenchendo valores ausentes na coluna 'Renda Mensal' com a média das rendas mensais
df_clientes['Renda Mensal'] = df_clientes['Renda Mensal'].fillna(df_clientes['Renda Mensal'].mean())
```

```
df_clientes['Renda Mensal'] = df_clientes['Renda Mensal'].fillna(df_clientes['Renda Mensal'])

df_clientes
```

	Nome	Idade	Renda Mensal	Limite de Crédito
0	João	25.0	5000.0	10000.0
1	Maria	30.0	6000.0	8000.0
2	Pedro	32.5	5250.0	8000.0
3	Ana	40.0	4500.0	12000.0
4	Mariana	35.0	5500.0	8000.0

Agora, vamos carregar os dados `gapminder`, que está no arquivo `gapminder.zip`.

```
gapminder = pd.read_csv("data/gapminder.zip", sep = "\t")
```

A função `head()` é usada para visualizar as primeiras linhas do conjunto de dados `gapminder`, oferecendo uma rápida visão geral da sua estrutura e conteúdo.

```
gapminder.head()
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

O método `info()` fornece informações sobre o conjunto de dados, incluindo o número de entradas, o tipo de dados de cada coluna e se há valores nulos.

```
gapminder.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
#   Column      Non-Null Count  Dtype
---  -
0   country     1704 non-null   object
1   continent   1704 non-null   object
```

```

2   year      1704 non-null   int64
3   lifeExp   1704 non-null   float64
4   pop       1704 non-null   int64
5   gdpPercap 1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB

```

A função `describe()` gera estatísticas descritivas para cada coluna numérica do conjunto de dados, como contagem, média, desvio padrão, mínimo e máximo.

```
gapminder.describe()
```

	year	lifeExp	pop	gdpPercap
count	1704.00000	1704.000000	1.704000e+03	1704.000000
mean	1979.50000	59.474439	2.960121e+07	7215.327081
std	17.26533	12.917107	1.061579e+08	9857.454543
min	1952.00000	23.599000	6.001100e+04	241.165876
25%	1965.75000	48.198000	2.793664e+06	1202.060309
50%	1979.50000	60.712500	7.023596e+06	3531.846988
75%	1993.25000	70.845500	1.958522e+07	9325.462346
max	2007.00000	82.603000	1.318683e+09	113523.132900

A função `value_counts()` conta o número de ocorrências de cada categoria na coluna “continent” do conjunto de dados `gapminder`, transforma os resultados em um `DataFrame`, renomeia as colunas para “continent” e “n” (indicando a contagem), e reconfigura o índice.

```
gapminder.value_counts("continent").to_frame("n").reset_index()
```

	continent	n
0	Africa	624
1	Asia	396
2	Europe	360
3	Americas	300
4	Oceania	24

No trecho abaixo, fazemos uma contagem de valores para as combinações únicas de categorias nas colunas “continent” e “year” do `DataFrame` `gapminder`. Os resultados são transformados em um `DataFrame`, renomeados como “continent”, “year” e “n” (indicando a contagem), e o índice é reconfigurado.

```
gapminder.value_counts(["continent", "year"]).to_frame("n").reset_index()
```

	continent	year	n
0	Africa	1952	52
1	Africa	1987	52
2	Africa	1957	52
3	Africa	2002	52
4	Africa	1997	52
5	Africa	1992	52
6	Africa	2007	52
7	Africa	1982	52
8	Africa	1977	52
9	Africa	1972	52
10	Africa	1967	52
11	Africa	1962	52
12	Asia	1952	33
13	Asia	2007	33
14	Asia	2002	33
15	Asia	1997	33
16	Asia	1992	33
17	Asia	1987	33
18	Asia	1977	33
19	Asia	1972	33
20	Asia	1967	33
21	Asia	1962	33
22	Asia	1957	33
23	Asia	1982	33
24	Europe	1982	30
25	Europe	1957	30
26	Europe	2007	30
27	Europe	2002	30
28	Europe	1997	30
29	Europe	1992	30
30	Europe	1987	30
31	Europe	1977	30
32	Europe	1972	30
33	Europe	1967	30
34	Europe	1962	30
35	Europe	1952	30
36	Americas	2002	25
37	Americas	2007	25
38	Americas	1952	25

```

39 Americas 1962 25
40 Americas 1967 25
41 Americas 1972 25
42 Americas 1977 25
43 Americas 1982 25
44 Americas 1987 25
45 Americas 1992 25
46 Americas 1997 25
47 Americas 1957 25
48 Oceania 1982 2
49 Oceania 2002 2
50 Oceania 1997 2
51 Oceania 1992 2
52 Oceania 1987 2
53 Oceania 1957 2
54 Oceania 1977 2
55 Oceania 1972 2
56 Oceania 1967 2
57 Oceania 1962 2
58 Oceania 1952 2
59 Oceania 2007 2

```

6.5 Dados organizados (tidy data)

Todas as tabelas abaixo tem o mesmo dado (foram tiradas do pacote `tidyr` do R), que mostra a quantidade de casos de uma doença e a população total de alguns países.

```

table1 = pd.read_csv("data/table1.csv")
table2 = pd.read_csv("data/table2.csv")
table3 = pd.read_csv("data/table3.csv")
table4a = pd.read_csv("data/table4a.csv")
table4b = pd.read_csv("data/table4b.csv")

```

```
table1
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898

```

4      China 1999 212258 1272915272
5      China 2000 213766 1280428583

```

```
table2
```

```

      country year      type      count
0  Afghanistan 1999      cases         745
1  Afghanistan 1999 population 19987071
2  Afghanistan 2000      cases         2666
3  Afghanistan 2000 population 20595360
4      Brazil 1999      cases        37737
5      Brazil 1999 population 172006362
6      Brazil 2000      cases        80488
7      Brazil 2000 population 174504898
8      China 1999      cases        212258
9      China 1999 population 1272915272
10     China 2000      cases        213766
11     China 2000 population 1280428583

```

```
table3
```

```

      country year      rate
0  Afghanistan 1999      745/19987071
1  Afghanistan 2000      2666/20595360
2      Brazil 1999      37737/172006362
3      Brazil 2000      80488/174504898
4      China 1999      212258/1272915272
5      China 2000      213766/1280428583

```

```
table4a
```

```

      country      1999      2000
0  Afghanistan      745      2666
1      Brazil      37737      80488
2      China      212258      213766

```

```
table4b
```

	country	1999	2000
0	Afghanistan	19987071	20595360
1	Brazil	172006362	174504898
2	China	1272915272	1280428583

O exemplo abaixo cria uma nova coluna chamada `rate` no DataFrame `table1`. A função `assign` adiciona uma nova coluna ao DataFrame, enquanto a expressão `lambda` calcula os valores para essa nova coluna.

```
table1.assign(rate = lambda _: 10000 * (_.cases / _.population))
```

	country	year	cases	population	rate
0	Afghanistan	1999	745	19987071	0.372741
1	Afghanistan	2000	2666	20595360	1.294466
2	Brazil	1999	37737	172006362	2.193930
3	Brazil	2000	80488	174504898	4.612363
4	China	1999	212258	1272915272	1.667495
5	China	2000	213766	1280428583	1.669488

No exemplo abaixo, agrupamos os dados do DataFrame `table1` pela coluna “year” (ano) e depois calcula a soma dos casos para cada ano. O método `groupby("year")` agrupa os dados por ano, criando grupos separados para cada ano. `as_index = False` especifica que a coluna usada para agrupamento (“year”) não deve ser definida como índice no DataFrame resultante. O método `agg` é usado para realizar uma operação de agregação nos grupos. Aqui, `np.sum` é usado para calcular a soma dos valores da coluna “cases” para cada grupo.

```
(table1.groupby("year", as_index = False)
    .agg(total_cases = ("cases", np.sum)))
```

	year	total_cases
0	1999	250740
1	2000	296920

Para fazer o mesmo com os dados da `table1`, temos que usar a função `pivot_table`:

```
table2_tidy = (table2.pivot_table(index = ["country", "year"], columns = "type", values =
    .reset_index()
    .rename_axis(None, axis = 1))

table2_tidy.assign(rate = lambda _: 10000 * (_.cases / _.population))
```

	country	year	cases	population	rate
0	Afghanistan	1999	745.0	1.998707e+07	0.372741
1	Afghanistan	2000	2666.0	2.059536e+07	1.294466
2	Brazil	1999	37737.0	1.720064e+08	2.193930
3	Brazil	2000	80488.0	1.745049e+08	4.612363
4	China	1999	212258.0	1.272915e+09	1.667495
5	China	2000	213766.0	1.280429e+09	1.669488

No exemplo acima, usamos o método `pivot_table` do pandas para reorganizar os dados do DataFrame `table2`. Ele reorganiza os dados de forma que os valores da coluna “count” sejam pivotados (transformados em colunas) com base nos valores únicos da combinação de “country” e “year”. Os parâmetros `index`, `columns` e `values` especificam respectivamente as colunas que serão usadas como índice, as que serão transformadas em colunas e os valores a serem preenchidos na tabela pivô. Após a operação de pivotagem, são encadeados métodos adicionais para modificar a estrutura do DataFrame resultante:

- `reset_index()` redefine os índices do DataFrame para índices numéricos padrão, movendo os índices anteriores (no caso, “country” e “year”) para colunas.
- `rename_axis(None, axis=1)` remove os nomes dos índices das colunas, substituindo-os por `None`. Isso é feito especificamente para limpar os nomes das colunas do DataFrame.

Após a transformação dos dados, a função `assign` é usada para criar uma nova coluna chamada `rate` no DataFrame resultante `table2_tidy`.

Agora, vamos fazer o mesmo para a `table4a` e `table4b`:

```
table4_tidy = (table4a.melt(id_vars = "country", value_vars = ["1999", "2000"], var_name =
                  .merge(table4b.melt(id_vars = "country", value_vars = ["1999", "2000"], va
                        on = ("country", "year")))

table4_tidy.assign(rate = lambda _: 10000 * (_.cases / _.population))
```

	country	year	cases	population	rate
0	Afghanistan	1999	745	19987071	0.372741
1	Brazil	1999	37737	172006362	2.193930
2	China	1999	212258	1272915272	1.667495
3	Afghanistan	2000	2666	20595360	1.294466
4	Brazil	2000	80488	174504898	4.612363
5	China	2000	213766	1280428583	1.669488

Os DataFrames `table4a` e `table4b` são derretidos usando o método `melt`:

- Para `table4a`, as colunas que permanecerão fixas são especificadas através do argumento `id_vars = "country"`, enquanto as colunas "1999" e "2000" são derretidas como variáveis usando `value_vars = ["1999", "2000"]`. Os nomes das variáveis derretidas são renomeadas para "year" e "cases" usando `var_name = "year"` e `value_name = "cases"`, respectivamente.
- Da mesma forma, para `table4b`, as colunas "country" e "1999", "2000" são derretidas, com os nomes das variáveis renomeadas para "year" e "population", respectivamente.

Os DataFrames resultantes do derretimento de `table4a` e `table4b` são mesclados usando o método `merge`. A mesclagem é feita com base nas colunas "country" e "year", garantindo que os dados correspondentes de `table4a` e `table4b` sejam combinados corretamente.

Finalmente, o método `assign` é usado para criar uma nova coluna chamada "rate", que representa a taxa de casos por 10.000 habitantes.

Para a `table3`, basta separar a coluna `cases` considerando o separador `\`:

```
print(table3)
```

	country	year	rate
0	Afghanistan	1999	745/19987071
1	Afghanistan	2000	2666/20595360
2	Brazil	1999	37737/172006362
3	Brazil	2000	80488/174504898
4	China	1999	212258/1272915272
5	China	2000	213766/1280428583

```
table3_tidy = (table3.assign(cases = lambda _: _.rate.str.split("/", expand = True)[0].astd
                           population = lambda _: _.rate.str.split("/", expand = True)[1].astd
                           .drop("rate", axis = 1))
```

```
table3_tidy
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

```
table3_tidy.assign(rate = lambda _: 10000 * (_.cases / _.population))
```

	country	year	cases	population	rate
0	Afghanistan	1999	745	19987071	0.372741
1	Afghanistan	2000	2666	20595360	1.294466
2	Brazil	1999	37737	172006362	2.193930
3	Brazil	2000	80488	174504898	4.612363
4	China	1999	212258	1272915272	1.667495
5	China	2000	213766	1280428583	1.669488

O parâmetro `expand` é utilizado no método `str.split()` para especificar se o resultado da divisão deve ser expandido em um DataFrame (se `True`) ou mantido como uma lista de valores (se `False`, que é o padrão).

Principais funcionalidades - ver aulas paulo

6.6 Visualização de dados

Duas bibliotecas amplamente utilizadas para visualização em Python são o `Matplotlib` e o `Plotnine`. O `Matplotlib` oferece uma ampla gama de opções para criar visualizações estáticas, desde gráficos simples até gráficos complexos e personalizados. Por outro lado, o `Plotnine` é uma biblioteca baseada na gramática de gráficos (parecido com o `ggplot2` do R), o que facilita a criação de visualizações elegantes e concisas usando uma sintaxe intuitiva e expressiva.

6.6.1 Matplotlib

Antes de começarmos a criar visualizações, é importante entender alguns conceitos básicos do `Matplotlib`:

- **Figura e Eixo (Axes):** No `Matplotlib`, uma figura é a janela ou página na qual tudo é desenhado. Dentro de uma figura, pode haver vários eixos (ou subplots), onde os dados são efetivamente plotados.
- **Método `plot()`:** O método `plot()` é usado para criar gráficos de linha, pontos ou marcadores. Ele aceita uma variedade de argumentos para personalizar a aparência do gráfico, como cor, estilo de linha, largura da linha, etc.
- **Customização:** O `Matplotlib` oferece muitas opções de personalização para ajustar a aparência dos gráficos, incluindo a adição de rótulos aos eixos, título do gráfico, legendas, entre outros.

Agora, vamos ver um exemplo de como criar um gráfico de pontos usando dados fictícios, onde cada unidade de dado está relacionada a uma empresa.

```
import matplotlib.pyplot as plt

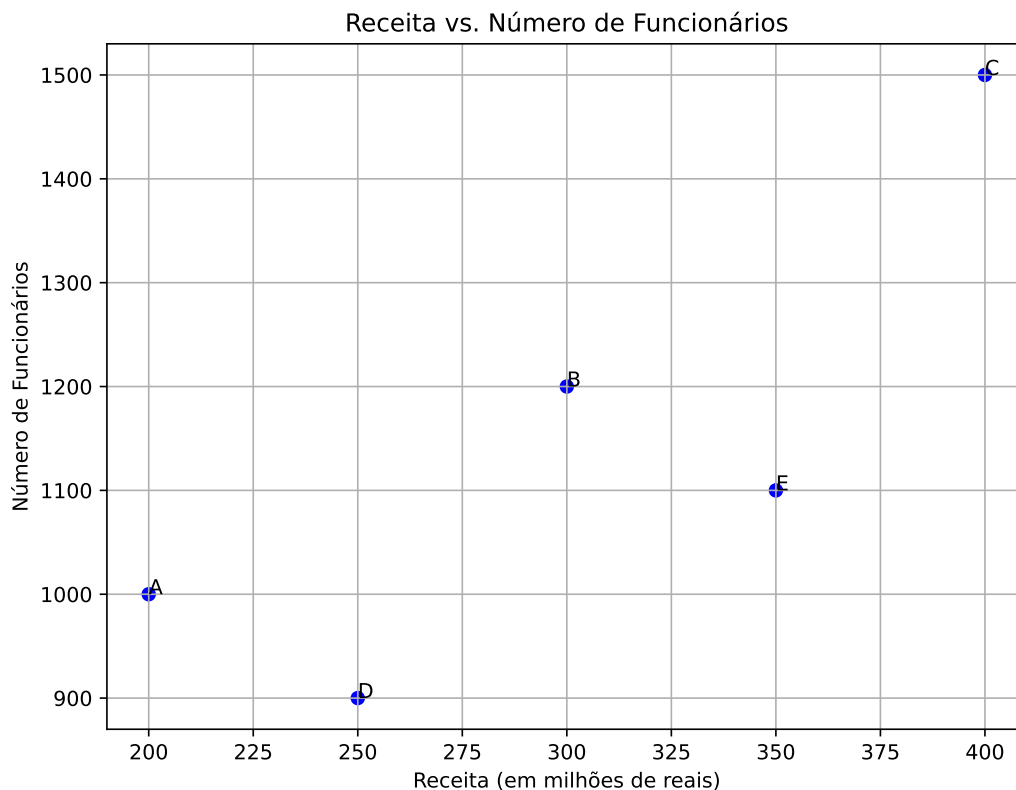
# Dados de exemplo: Nome das empresas, receita e número de funcionários
empresas = ['A', 'B', 'C', 'D', 'E']
receita = [200, 300, 400, 250, 350] # em milhões de reais
funcionarios = [1000, 1200, 1500, 900, 1100]

# Criando o gráfico de pontos
plt.figure(figsize=(8, 6))
plt.scatter(receita, funcionarios, color='blue', marker='o')

# Adicionando rótulos e título
plt.xlabel('Receita (em milhões de reais)')
plt.ylabel('Número de Funcionários')
plt.title('Receita vs. Número de Funcionários')

# Adicionando anotações para cada ponto
for i in range(len(empresas)):
    plt.annotate(empresas[i], (receita[i], funcionarios[i]))

# Exibindo o gráfico
plt.grid(True)
plt.show()
```



Neste exemplo, cada ponto no gráfico representa uma empresa, onde o eixo x representa a receita (em milhões de reais) e o eixo y representa o número de funcionários. As anotações são usadas para identificar cada empresa no gráfico.

Na sequência, utilizamos o Matplotlib para criar um gráfico de linha que representa a evolução das vendas de dois produtos ao longo de vários anos. Cada ponto no gráfico representa o número de vendas em um ano específico.

```
import matplotlib.pyplot as plt

# Dados de exemplo: Anos e vendas de produtos
anos = [2010, 2011, 2012, 2013, 2014, 2015, 2016]
vendas_produto_A = [500, 600, 550, 700, 800, 750, 900]
vendas_produto_B = [400, 450, 500, 550, 600, 650, 700]

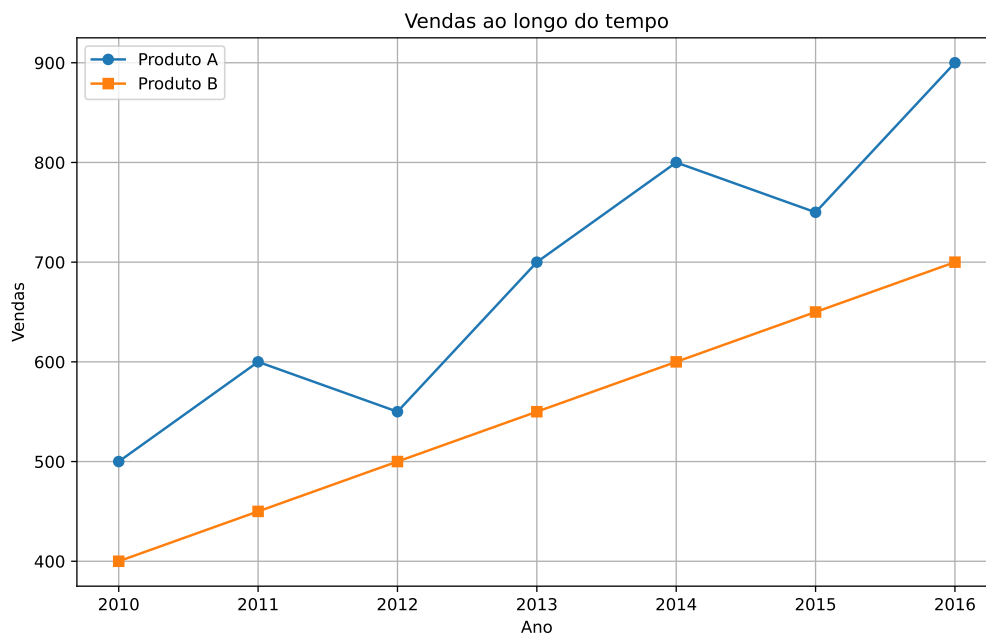
# Criando o gráfico de linha
```

```
plt.figure(figsize=(10, 6))
plt.plot(anos, vendas_produto_A, marker='o', label='Produto A')
plt.plot(anos, vendas_produto_B, marker='s', label='Produto B')

# Adicionando rótulos e título
plt.xlabel('Ano')
plt.ylabel('Vendas')
plt.title('Vendas ao longo do tempo')

# Adicionando legenda
plt.legend()

# Exibindo o gráfico
plt.grid(True)
plt.show()
```



6.6.2 Plotnine

Plotnine é uma biblioteca em Python que permite criar visualizações de dados estatísticos de uma forma simples e concisa, utilizando a gramática de gráficos do R (também conhecida

como ggplot2). Essa gramática consiste em uma abordagem declarativa para a construção de gráficos, onde os elementos visuais são adicionados em camadas para formar o gráfico final.

```
from plotnine import *
```

Para exemplificar, vamos utilizar a base de dados `gapminder`.

```
((ggplot(gapminder, aes(x = "continent", fill = "continent"))) +  
  geom_bar(aes(y = "stat(count) / 12"), alpha = 0.75) +  
  labs(x = "", y = "Number of countries", title = "Continents") +  
  theme(legend_position = "none") +  
  coord_flip()+  
  theme_bw())  
  .show())
```

```
((ggplot(gapminder, aes(x = "lifeExp", y = "stat(density)"))) +  
  geom_histogram(fill = "blue", color = "white", alpha = 0.5) +  
  labs(x = "Life Expectancy", y = "", title = "Gapminder"))  
  .show())
```

```
((ggplot(gapminder, aes(x = "lifeExp", y = "stat(density)"))) +  
  geom_histogram(fill = "blue", color = "white", alpha = 0.5) +  
  labs(x = "Life Expectancy", y = "", title = "Gapminder") +  
  facet_wrap("~ continent", nrow = 1) +  
  theme(figure_size = (12, 2))).  
  show)
```

```
((gapminder.groupby(["continent", "year"], as_index = False)  
  .agg(median_lifeExp = ("lifeExp", np.median))  
  .pipe(lambda _: ggplot(_, aes(x = "year", y = "median_lifeExp", color = "continent"))  
    geom_line(size = 0.75) +  
    geom_point(size = 1.5) +  
    labs(x = "Year", y = "Median Life Expectancy", color = "Continen")  
    .show())
```

Parte III

Estudo de casos

7 Estudo de casos práticos

O objetivo deste capítulo é apresentar estudos de casos reais usando tanto **R** quanto **Python**.

! Importante

Página ainda em construção!

References

- McKinney, Wes. 2022. *Python for data analysis*. " O'Reilly Media, Inc."
- Rosling, Hans. 2012. «Data - Gapminder.org». <http://www.gapminder.org/data/>.
- Wilkinson, Leland. 2012. *The grammar of graphics*. Springer.