

R para Análises Científicas Reproduzíveis

Introdução ao R e ao Rstudio

Saulo Machado Jacques

Esse trabalho pode ser copiado, compartilhado e modificado desde que citada a fonte de acordo com a Licença Creative Commons - Atribuição 4.0 Internacional.

Introdução ao RStudio

- Apresentação inicial
- Perguntar quantas pessoas estão familiarizadas com o R
- Conferir se o RStudio está instalado em todos os computadores
- Confirmar a instalação do `ggplot2` e do `gapminder`

Hoje iremos aprender sobre:

- Apresentação do RStudio
- Apresentação do R
- Como ler os dados com R
- Criando gráficos

Layout Básico do RStudio

Quando abrimos o RStudio, são apresentados três painéis:

- Console Interativo do R (área esquerda inteira)
- Environment/History (Quadro superior direito)
- Files/Plots/Packages/Help (aba no quadro inferior direito)

Uma vez que arquivos como script do R é aberto, o painel de script também abrirá na parte superior à esquerda.

Trabalhando dentro do RStudio

Inicie escrevendo em um arquivo .R e use comando/ atalho do Rstudio para “jogar” a linha atual, selecionada ou modificada para o console interativo do R.

Dica: Enviando para o console interativo R

Para executar a linha atual, clique no botão ‘Run’, logo acima do painel de arquivos. Ou use o atalho que pode ser visto passando o mouse sobre o botão.

Para executar um bloco de código, selecione-o e depois pressione ‘Run’. Se você tiver modificado uma linha do código dentro de um bloco de código que você acabou de executar. Não há necessidade de voltar a selecionar a seção e Run, você pode usar o próximo botão ao lado, `Re-run the previous region`. Isto executará o bloco de código anterior incluindo as modificações que você fez.

Introdução a R

O Console R

Pode ser um ambiente útil para testar ideias antes de adicioná-las a um arquivo de script R. Este console no RStudio é o mesmo que você teria se você digitasse ‘R’ em seu ambiente de linha de comando. A primeira coisa que você vai ver na sessão interativa R é um monte de informações, seguido por um “>” e um cursor intermitente. Você digita comandos, R tenta executá-los e, em seguida, retorna um resultado.

Usando R como uma calculadora

A coisa mais simples que você poderia fazer com R é fazer aritmética:

```
1 + 100
```

```
[1] 101
```

E o R vai imprimir a resposta, com um precedente “[1]”. Não se preocupe com isso por agora, vamos explicar isso mais tarde. Por agora pense nisso como indicando saída.

Assim como o bash, se você digitar um comando incompleto, o R esperará que você o complete:

```
> 1 +
```

```
+
```

Sempre que pressionamos o “ENTER” e a sessão R mostrar um “+” em vez de um “>”, significa que está esperando que o comando seja concluído. Se quisermos cancelar um comando, basta clicar em “Esc” e o RStudio volta ao prompt “>”.

Dica: Cancelar comandos

Se você estiver usando R a partir da linha de comando em vez de dentro RStudio, você precisará usar **Ctrl+C** em vez de **Esc** para cancelar o comando. Isso também se aplica aos usuários de Mac!

Cancelar um comando não é útil apenas para terminar comandos incompletos: você também pode usá-lo para informar ao R para parar a execução de um código (por exemplo, se estiver demorando muito mais do que você esperava), ou para se livrar do código que você está escrevendo atualmente.

Ao usar R como uma calculadora, a ordem das operações é a mesma que aprendemos na escola.

Da precedência mais alta a mais baixa:

- Parênteses: (,)
- Exponentes: \wedge ou $**$
- Dividir: /
- Multiplicar: *
- Adicionar: +
- Subtrair: -

```
3 + 5 * 2
```

```
[1] 13
```

Use parênteses no grupo para forçar a ordem de avaliação se ela difere do padrão, ou para definir sua própria ordem.

```
(3 + 5) * 2
```

```
[1] 16
```

Mas isso pode ficar complicado quando é desnecessário:

```
(3 + (5 * (2 ^ 2))) # difícil de ler  
3 + 5 * 2 ^ 2       # Mais fácil de ler, uma vez que você sabe as regras  
3 + 5 * (2 ^ 2)     # Se você esquecer algumas regras, isso pode ajudar
```

O texto digitado após cada linha de código é chamado de comentário. Qualquer coisa que se segue após a cerquilha (ou hash) `#` é ignorado pelo R quando for executar o código.

Números muito pequenos ou muito grandes obtêm uma notação científica:

```
2/10000
```

```
[1] 2e-04
```

Isso é uma abreviação para “multiplicado por 10^{XX} ”. Assim `2e-4` é abreviação para `2 * 10^(-4)`.

Podemos escrever números em notação científica:

```
5e3 # Observe a ausência do sinal negativo aqui
```

```
[1] 5000
```

Funções matemáticas

R possui muitas funções matemáticas embutidas. Para usar (“chamar”) uma função, simplesmente digite seu nome, seguido por parênteses. Qualquer coisa que digitemos dentro desses parênteses é chamado de argumentos da função:

```
sin(1) # funções trigonométricas
```

```
[1] 0.841471
```

```
log(1) # Logaritmo natural
```

```
[1] 0
```

```
log10(10) # logaritmo de base -10
```

```
[1] 1
```

```
exp(0.5) # e^(1/2)
```

```
[1] 1.648721
```

Não se preocupe em tentar lembrar cada função em R. Você pode simplesmente procurá-los no Google, ou se você lembrar do início do nome da função, use tab para completar RStudio. Esta é uma das vantagens que RStudio tem sobre R: existem habilidades de autocompletar que permitem procurar mais facilmente por funções, seus argumentos e os valores contidos nelas. Digitando `?` Antes do nome de um comando abre a página de ajuda para esse comando. Além de fornecer uma descrição detalhada do comando e como ele funciona, a rolagem na parte inferior da página exibe uma coleção de exemplos de códigos que ilustram o uso dos comandos. Passaremos por um exemplo mais adiante.

Fazendo Comparações

Podemos também fazer comparação em R:

```
1 == 1 # Igualdade (note dois sinais iguais, lido como "é igual a")
```

```
[1] TRUE
```

```

1 != 2 # diferença (lido como "não é igual a")
[1] TRUE
1 < 2 # menor que
[1] TRUE
1 <= 1 # menor ou igual a
[1] TRUE
1 > 0 # maior que
[1] TRUE
1 >= -9 # maior ou igual a
[1] TRUE

```

Dica: Comparando Números

Uma observação sobre comparação de números: você nunca deve usar `==` para comparar dois números a menos que sejam inteiros.

Computadores só podem representar números decimais com um certo grau de precisão, então dois números que parecem os mesmos quando impressos pelo R podem realmente ter diferentes representações subjacentes e portanto ser diferente por uma pequena margem de erro (chamada tolerância numérica da máquina).

Em vez disso, você deve usar a função `all.equal`. Mais informações: <http://floating-point-gui.de/>

Variáveis e atribuição

Podemos armazenar valores em variáveis usando o operador de atribuição `<-`, assim:

```
x <- 1/40
```

Note que a atribuição não imprime um valor. Em vez disso, nós armazenamos isso para mais tarde em algo chamado **variable**. `x` agora contém o **value** 0.025:

```
x
```

```
[1] 0.025
```

Mais precisamente, o valor armazenado é uma *aproximação decimal* dessa fração chamada de número de ponto flutuante.

Procure a aba **Environment** em um dos painéis do RStudio. É possível ver que `x` e o seu valor apareceram lá. A variável `x` criada pode ser usada no lugar de um número em qualquer cálculo que espera um número:

```
log(x)
```

```
[1] -3.688879
```

Observe também que podemos sobrescrever o valor das variáveis:

```
x <- 100
```

`x` continha o valor 0.025 e agora tem o valor 100.

Nomes de variáveis podem conter letras, números, underscores e pontos. Eles não podem começar com um número nem conter espaços. Diferentes pessoas usam convenções diferentes para nomes de variáveis longas. Isso inclui:

- ponto.entre.palavras
- underscores_entre_palavras
- IniciaisMaiusculasEntrePalavras

A tecla tab pode ser usada para autocompletar. Importante: O que você usa é decisão sua, mas **seja coerente**. Nomes das variáveis precisam ser informativos.

Dica: Avisos vs. Erros

Preste atenção quando R faz algo inesperado! Erros, como acima, são lançados quando R não pode proceder com um cálculo. Os avisos, por outro lado, normalmente significam que a função foi executada, mas provavelmente não funcionou como esperado. Em ambos os casos, a mensagem que R imprime geralmente fornece dicas sobre como corrigir um problema.

Comparando coisas - O retorno

Mostre que você pode comparar uma variável com um número!

Desafio 1

Execute o código abaixo e escreva um comando para comparar a massa com a idade. A massa é maior que a idade?

```
massa <- 47.5
idade <- 122
massa <- massa * 2.3
idade <- idade - 20
```

Layout do projeto

O processo científico é naturalmente incremental, e muitos projetos se iniciam como notas aleatórias, alguns códigos, então um manuscrito e, finalmente, é um pouco disso tudo combinado. Muitas pessoas tendem a organizar seus projetos dessa forma:

Há muitas razões pelas quais devemos *SEMPRE* evitar isso::

1. É muito difícil dizer qual versão de seus dados é a original e qual é a modificada;
2. Fica muito confuso porque mescla arquivos com várias extensões;
3. Provavelmente leva muito tempo para realmente encontrar coisas, e relacionar as figuras corretas para o exato código que foi usado para gerá-lo;

Um bom layout de projeto, em última análise, tornar sua vida mais fácil:

- Irá ajudar a garantir a integridade dos seus dados;
- Torna mais simples compartilhar seu código com outra pessoa (um companheiro de laboratório, colaborador ou supervisor);
- Permite fazer mais facilmente o upload seu código com a submissão do seu manuscrito;
- Torna mais fácil retomar um projeto após uma pausa.

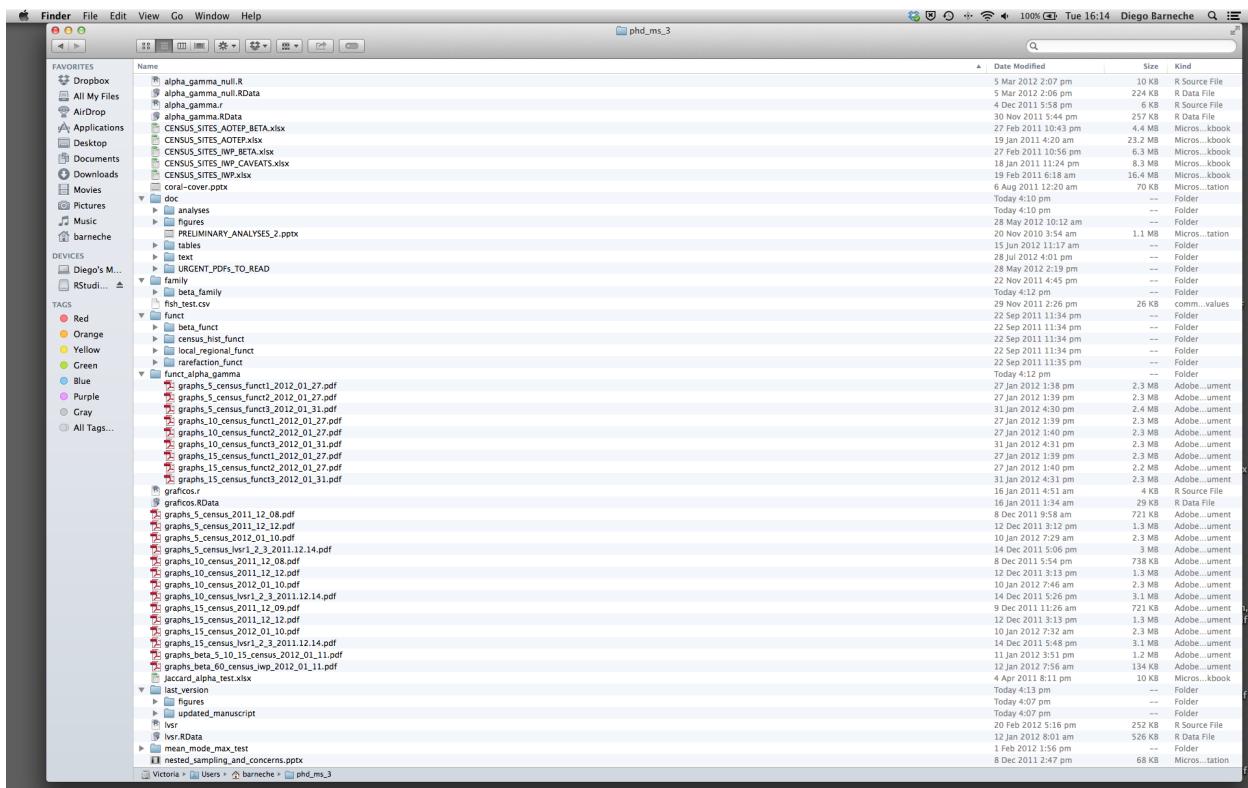


Figure 1:

Uma possível solução

Felizmente, existem ferramentas e pacotes que podem ajudá-lo a gerenciar seu trabalho de forma eficaz! E um dos aspectos mais poderosos e úteis do RStudio é a sua funcionalidade de gerenciamento de projetos. O passo seguinte é criar um projeto autônomo e reproduzível.

Desafio: criando um projeto autônomo

Passos para criar um projeto no RStudio:

1. Clique no botão de menu “File”, depois em “New Project”.
2. Clique em “New Directory”.
3. Clique em “Empty Project”.
4. Digite o nome do diretório para armazenar seu projeto, ex. “meu_projeto”.
5. Certifique-se de que a caixa de verificação para “Create a git repository” está selecionada.
6. Clique no botão “Create Project”.

Quando começamos R neste diretório do projeto, ou abrimos esse projeto com o RStudio, todo nosso trabalho neste projeto será inteiramente autônomo neste diretório.

Melhores práticas para organização de projetos

Embora não haja uma “melhor” maneira de elaborar um projeto, existem alguns princípios a adotar que tornarão a gestão de projetos mais fácil:

Tratar dados como somente leitura

Este é provavelmente o objetivo mais importante da criação de um projeto. Os dados tipicamente demandam tempo e dinheiro para se obter. Trabalhando com eles interativamente (por exemplo, no Excel) onde eles podem ser modificados significa que nunca teremos certeza de onde os dados vieram, ou se e como eles foram modificados desde a coleta. Por isso é uma boa ideia tratar seus dados como “somente leitura”.

“Limpeza” de dados

Em muitos casos, seus dados estão “sujos”: Será necessário um pré-processamento profundo adequá-los a um formato R (ou qualquer outra linguagem de programação). Esta etapa é às vezes chamada de “data munging” ou “data wrangling”. Uma medida útil é armazenar esses dados brutos originais em uma pasta separada e criar uma segunda pasta de dados “somente leitura” para conjuntos de dados “limpos”.

Tratar seus resultados das análises (output) como dispensáveis

Qualquer coisa gerada por seus scripts deve ser tratada como dispensáveis. Isso quer dizer que todos os resultados devem ser possíveis de serem produzidos novamente a partir de seus scripts. Existem diferentes formas para gerenciar essa saída. Uma forma útil é ter uma pasta de output com diferentes subdiretórios para cada análise. Isso facilita mais tarde, ainda mais quando muitas das análises são exploratórias e não acabam sendo utilizadas no projeto final, mas algumas das análises são compartilhadas entre projetos.

Dica: ProjectTemplate - uma possível solução

Uma maneira de automatizar o gerenciamento de projetos é instalar o pacote de terceiros, **ProjectTemplate**. Este pacote configurará uma estrutura de diretório ideal para o gerenciamento de projetos. Isso é muito útil, pois permite que você tenha sua análise / fluxo de trabalho organizados e estruturados. Com a funcionalidade padrão do projeto RStudio e o Git, é possível acompanhar o trabalho, bem como ser capaz de compartilhá-lo com outros colaboradores.

1. Instalar o **ProjectTemplate**.
2. Carregar a biblioteca
3. Iniciar o projeto:

```
install.packages("ProjectTemplate")
library(ProjectTemplate)
create.project("../my_project", merge.strategy = "allow.non.conflict")
```

Para obter mais informações sobre ProjectTemplate e sua funcionalidade, visite a > home page ProjectTemplate

Organizar funções e aplicações separadamente

Quando o projeto é recém-iniciado, o arquivo de script geralmente contém muitas linhas do código executado diretamente. À medida que amadurece, os pedaços reutilizáveis são puxados para próprias funções. Uma boa medida é organizá-los em pastas separadas; uma para armazenar as funções úteis que serão reutilizadas em análises e projetos, e uma para armazenar os scripts de análise.

Dica: evitando a duplicação

Muitas vezes você pode estar em meio a trabalhos com dados ou scripts de análise em vários projetos. Normalmente, você quer evitar a duplicação para economizar espaço e evitar fazer atualizações de código em vários lugares.

Neste caso, o útil fazer “links simbólicos”, que são basicamente atalhos para arquivos em outro lugar em um sistema de arquivos. No Linux e OS X você pode usar o comando `ln -s` e, nas janelas, você pode criar um atalho ou usar o comando `mklink` do terminal do windows.

Salva os dados no diretório de dados

Agora que temos uma boa estrutura de diretórios, alocaremos/salvaremos o arquivo de dados no diretório `data/`.

Desafio 1

Faça o download dos dados gapminder aqui.

1. Use o `Download ZIP` a última opção, localizado no menu do lado direito. Para fazer download do arquivo `.zip` para
2. Sua pasta de downloads.
3. Descompacte o arquivo.
4. Crie um diretório de dados dentro do seu projeto
5. Mova o arquivo para o `data/` dentro do seu projeto.

Carregaremos e inspecionaremos esses dados posteriormente.

Controle de versão

Também criamos nosso projeto de integração com o GIT, colocando-o sob controle de versão. O RStudio tem uma interface mais “amigável” para o git do shell, mas é muito limitado no que pode fazer, então ocasionalmente é necessário usar o shell. Vamos fazer um commit inicial de nossos arquivos de modelo. O painel `Workspace/History` tem uma guia para “Git”, onde podemos gerenciar cada arquivo: Um “A” verde é encontrado ao lado de arquivos e pastas de estágio e pontos de interrogação amarelos próximo aos arquivos ou pastas Git ainda desconhecidos. O RStudio também mostra bem a diferença entre arquivos de diferentes commits.

Dica: criando versão de resultado dispensável

Geralmente não interessa manter versão do resultado dispensável (ou dados somente leitura). Para isso, modificamos a opção de arquivo `.gitignore` informando ao Git para ignorar esses arquivos e diretórios.

Desafio 2

1. Crie um diretório dentro do seu projeto chamado `graphs`.
2. Modifique o arquivo `.gitignore` para conter `graphs /` para que não seja criada uma versão desta saída dispensável.

Adicione as pastas recém-criadas ao controle de versão usando A interface git.

Tipos de dados

R tem 5 tipos atômicos básicos (o que significa que eles não podem ser divididos em qualquer coisa menor):

- Lógico (por exemplo, TRUE,FALSE)
- Numérico
- Inteiro (por exemplo, 2L,as.integer (3))
- Decimal (“double”)
- Complexo (isto é, números complexos) (por exemplo, “1 + 0i”, “1 + 4i”)
- Text (“character”) (por exemplo, a, " swc ' ',Leve-me ao Seu Líder“)

O tipo numérico padrão é “decimal”, que pode ser convertido para inteiro. Os inteiros ocupam menos espaço e consomem menos memória nas análises. Podemos armazenar qualquer um desses tipos de dados dentro de uma variável. Se não tivermos a certeza do tipo de dados que possuímos, existem várias funções que podem ser utilizadas para descobrir:

```
typeof() # qual é o seu tipo atômico?  
is.logical() # é dado TRUE / FALSE?  
is.numeric() # é numérico?  
is.integer() # é um inteiro?  
is.complex() # é número complexos?  
is.character() # são dados de caracteres?
```

Desafio 1: Tipos de dados

Use seu conhecimento de como atribuir um valor a uma variável para criar exemplos de dados com as seguintes características:

- 1) Nome da variável: ‘resposta’, Tipo: lógico
- 2) Nome da variável: ‘altura’, Tipo: numérico
- 3) Nome da variável: ‘sobrenome’, Tipo: caracter

Para cada variável que você criou, teste se apresentam o tipo de dados pretendido. Econtrou algo inesperado?

Estruturas de dados

Foi abordado como trabalhar com um único número. E se houver mais de um? Uma variável que pode conter mais de um item é chamada de estrutura de dados. Existem cinco estruturas de dados que normalmente encontraremos no R:

- Vetor
- Fator
- Lista
- Matriz
- data.frame Por enquanto abordaremos os vetores com mais detalhes, para descobrir mais sobre os tipos de dados.

Vetores

Vetor pode ser visto como uma longa lista de dados separados por vírgulas. É o tipo de dados mais importante, pois todos os outros tipos são baseados em vetores. Importante: eles só podem conter um tipo de dados. Um vetor pode conter qualquer um dos cinco tipos que apresentamos antes:

- Lógico (TRUE, FALSE)

- Inteiro (2L, `as.integer(3)`)
- Numérico (real ou decimal) (2, 2.0, pi)
- Complexo (1 + 0i, 1 + 4i)
- Caractere ("a", "swc") Criando um vetor vazio com `vector()` ou usando a função concatenada `c()`.

```
x <- vector()
x
```

```
logical(0)
```

Por padrão, ele cria um vetor vazio (isto é, um comprimento igual a 0) de tipo “lógico”.

```
x <- vector(length = 10) # com um comprimento predefinido
x
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Nesse caso, o número de FALSEs gerado deve ser igual a 10. Que podemos posteriormente especificar o tipo atômico que queremos usar.

```
x <- vector("character", length = 10) # com comprimento e tipo predefinidos
x
```

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

Podemos também usar a função concatenar para combinar quaisquer valores que quisermos em um vetor (desde que sejam do mesmo tipo atômico!).

```
x <- c(10, 12, 45, 33)
x
```

```
[1] 10 12 45 33
```

Também podemos criar vetores como sequência de números das seguintes formas:

```
series <- 1:10
series
```

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

```
seq(10)
```

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

```
seq(1, 10, by = 0.1)
```

```
[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3
[15] 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7
[29] 3.8 3.9 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.0 5.1
[43] 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9 6.0 6.1 6.2 6.3 6.4 6.5
[57] 6.6 6.7 6.8 6.9 7.0 7.1 7.2 7.3 7.4 7.5 7.6 7.7 7.8 7.9
[71] 8.0 8.1 8.2 8.3 8.4 8.5 8.6 8.7 8.8 8.9 9.0 9.1 9.2 9.3
[85] 9.4 9.5 9.6 9.7 9.8 9.9 10.0
```

Podemos também usar a função concatenar para adicionar elementos a um vetor:

```
x <- c(x, 57)
x
```

```
[1] 10 12 45 33 57
```

Se combinarmos vários tipos, o R converterá para o tipo mais simples. Isso é chamado de coerção implícita. A regra de coerção segue a ordem `logical -> integer -> numeric -> complex -> character`.

Vetores também podem ser forçados explicitamente usando o `as. <class_name>`. Um exemplo:

```
as.numeric()  
  
numeric(0)  
  
as.character()
```

```
character(0)
```

O R tentará fazer o que tem mais sentido para esse valor:

```
as.character(x)  
  
[1] "10" "12" "45" "33" "57"  
  
as.complex(x)  
  
[1] 10+0i 12+0i 45+0i 33+0i 57+0i  
  
x <- 0:6  
  
as.logical(x)  
  
[1] FALSE TRUE TRUE TRUE TRUE TRUE
```

Esse é um “comportamento” que encontraremos em muitas linguagens de programação. 0 é tratado como FALSE, enquanto todos os outros números são tratados como TRUE. Às vezes, as coerções (conversão dos tipos de dados), especialmente as que não fazem sentido, não funcionam.

Em alguns casos, R não será capaz de resolver com algo que faça sentido:

```
x <- c("a", "b", "c")  
as.numeric(x)  
  
Warning: NAs introduced by coercion  
  
[1] NA NA NA  
as.logical(x)  
  
[1] NA NA NA
```

Em ambos os casos, retornou um vetor de “NAs”, e no primeiro caso também retornou uma mensagem de aviso.

Dica: Objetos Especiais

“NA” é um objeto especial no R que denota um valor ausente. NA pode ocorrer em qualquer tipo de vetor. Existem alguns outros tipos de objetos especiais: Inf denota infinito (pode ser positivo ou negativo), Enquanto NaN significa que não é um número, um valor indefinido (ou seja, 0 / 0). NULL indica que a estrutura de dados não existe.

Podemos fazer perguntas sobre a estrutura dos vetores:

```
x <- 0:10  
tail(x, n=2) # obter os últimos 'n' elementos  
  
[1] 9 10  
  
head(x, n=1) # obter os primeiros 'n' elementos  
  
[1] 0
```

```

length(x)

[1] 11

str(x)

int [1:11] 0 1 2 3 4 5 6 7 8 9 ...

```

Podemos nomear os vetores:

```

x <- 1:4
names(x) <- c("a", "b", "c", "d")
x

a b c d
1 2 3 4

```

Matrizes

Outra estrutura de dados que encontraremos são matrizes. Na verdade as matrizes são apenas vetores atômicos, com atributos de dimensão adicionados. Podemos criar uma matriz utilizando a função `matrix`. A seguir vamos gerar alguns dados aleatórios:

```

set.seed(1) # garante que os números aleatórios serão os mesmos independente se rodados novamente.
x <- matrix(rnorm(18), ncol=6, nrow=3)
x

[,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,] -0.6264538  1.5952808 0.4874291 -0.3053884 -0.6212406 -0.04493361
[2,]  0.1836433  0.3295078 0.7383247  1.5117812 -2.2146999 -0.01619026
[3,] -0.8356286 -0.8204684 0.5757814  0.3898432  1.1249309  0.94383621

str(x)

num [1:3, 1:6] -0.626 0.184 -0.836 1.595 0.33 ...

```

Podemos usar `rownames`, `colnames` e `dimnames` para definir ou recuperar a coluna e `rownames` de uma matriz. As funções `nrow` e `ncol` informará o número de linhas e colunas (isso também se aplica aos data frames!), enquanto `length` irá dizer-lhe o número de elementos.

Desafio 3

Qual você acha que será o resultado de obtido a partir de `Length (x)`? Tente! Você estava certo? Consegue explicar o esperado e o resultado encontrado?

Desafio 4

Faça outra matriz, desta vez contendo os números 1:50, Com 5 colunas e 10 linhas. A função `matrix` completa a sua matriz por coluna, ou por linha, como seu modo padrão? Veja se você pode descobrir como mudar isso. (Dica: leia a documentação para `matrix!`)

Fatores

Os fatores são vetores especiais que representam dados categóricos. Fatores podem ser ordenados ou não ordenados e são importantes para funções de modelos como `Aov ()`, `lm ()` e `glm ()` e também em métodos de plotagem.

Os fatores só podem conter valores predefinidos, e podemos criar um com a função `factor`:

```
x <- factor(c("yes", "no", "no", "yes", "yes"))
x
```

```
[1] yes no  no  yes yes
Levels: no yes
```

Podemos ver que a saída é muito semelhante a um vetor de caracteres, mas com um componente de níveis anexado. Isso fica mais claro quando olhamos para a sua estrutura:

```
str(x)
```

```
Factor w/ 2 levels "no","yes": 2 1 1 2 2
```

Isso revela algo importante: os fatores parecem (e muitas vezes se comportam) vectores de caracteres, mas, na verdade, eles são inteiros, e aqui podemos ver que “no” é representado por um “1”, e “yes” representado por um “2”.

Nas funções de modelagem, é importante saber quais são os níveis “basais”. Por padrão, a ordenação é determinada por ordem alfabética das palavras inseridas. Isso pode ser alterado especificando os níveis, da seguinte forma:

```
x <- factor(c("case", "control", "control", "case"), levels = c("control", "case"))
str(x)
```

```
Factor w/ 2 levels "control","case": 2 1 1 2
```

Neste caso explicitamos ao R que “control” deve ser representado por 1, e “Case” por 2. Esta designação pode ser muito importante para a interpretação dos resultados de modelos estatísticos!

Listas

Se quisermos combinar diferentes tipos de dados, precisaremos usar listas. As listas atuam como “contêineres” e podem conter qualquer tipo de estrutura de si mesmos! As listas podem ser criadas usando `list` ou convertidas de outros objetos usando `as.list ()`:

```
x <- list(1, "a", TRUE, 1+4i)
x
```

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

```
[[2]]
[1] "a"
```

```
[[3]]
[1] TRUE
```

```
[[4]]
[1] 1+4i
```

Cada elemento da lista é denotado por um [[nos resultados (output). Dentro de cada elemento de lista está um vetor atômico de comprimento um.

As listas também podem conter objetos mais complexos:

```
xlist <- list(a = "Research Bazaar", b = 1:10, data = head(iris))
xlist

$a
[1] "Research Bazaar"

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

$data
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1       3.5        1.4       0.2   setosa
2          4.9       3.0        1.4       0.2   setosa
3          4.7       3.2        1.3       0.2   setosa
4          4.6       3.1        1.5       0.2   setosa
5          5.0       3.6        1.4       0.2   setosa
6          5.4       3.9        1.7       0.4   setosa
```

Nesse caso, nossa lista contém um vetor de caracteres de comprimento um, um vetor numérico com 10 entradas e um pequeno data frame de um dos muitos conjuntos de dados pré-carregados de R (ver `?data`). Também demos um nome para cada elemento da lista, razão pela qual vemos `$a` em vez de `[[1]]`.

Desafio 5

Crie uma lista de comprimento dois contendo um vetor de caracteres para as seguintes seções:

- Tipos de dados
- Estruturas de dados

Preencha cada vetor de caracteres com os nomes dos tipos de dados e estruturas de dados vistas até agora.

Listas são extremamente úteis dentro das funções. Podemos “grampear” um grupo de diferentes tipos de resultados em um único objeto que uma função pode retornar. Na verdade muitas funções de R que retornam resultados complexos armazenam seus resultados em uma lista.

Data frames

Os data frames são semelhantes a matrizes, exceto que cada coluna pode ser um tipo atômico diferente. Quando olhado em detalhes, os data frames são, na verdade, listas onde cada elemento é um vetor atômico, com a restrição adicional de que todos têm o mesmo comprimento. Se retirarmos uma coluna de um data frame, teremos um vetor.

Os data frames podem ser criados manualmente com a função `data.frame`:

```
df <- data.frame(id = c('a', 'b', 'c', 'd', 'e', 'f'), x = 1:6, y = c(214:219))
df

  id x     y
1  a 1 214
2  b 2 215
3  c 3 216
4  d 4 217
5  e 5 218
```

```
6 f 6 219
```

Data frames são bem legais, pois parecem muito com uma tabela de dados que podemos armazenar os dados.

Desafio: Data frames

Tente usar a função `length` para consultar o data frame `df`. O resultado foi o que esperava?

Cada coluna no data frame é apenas um elemento de lista, razão pela qual quando pedimos pelo `length` do data frame, ele informa o número de colunas. Se realmente quisermos saber o número de linhas, podemos usar a função `nrow`. Podemos adicionar linhas ou colunas a um data.frame usando `rbind` ou `cbind` (estes são os equivalentes bidimensionais da função `c`):

```
df2 <- rbind(df, df)
df2
```

```
  id x  y
1  a 1 214
2  b 2 215
3  c 3 216
4  d 4 217
5  e 5 218
6  f 6 219
7  a 1 214
8  b 2 215
9  c 3 216
10 d 4 217
11 e 5 218
12 f 6 219
```

```
df3 <- cbind(df, df)
df3
```

```
  id x  y id x  y
1  a 1 214  a 1 214
2  b 2 215  b 2 215
3  c 3 216  c 3 216
4  d 4 217  d 4 217
5  e 5 218  e 5 218
6  f 6 219  f 6 219
```

Desafio 1

Crie um data frame que contenha as seguintes informações:

- Nome próprio
- Sobrenome
- Idade

Em seguida, use `rbind` para adicionar as mesmas informações para as pessoas sentadas perto de você.

Agora use `cbind` para adicionar uma coluna de elementos lógicos respondendo à pergunta, “Há alguma coisa ainda confusa nas operação o R até aqui?”

Realizando “Subsetting”

Subsetting é o processo de subamostrar os dados selecionando apenas um grupo específico. Aqui usaremos a expressão subset, para o processo, ou subsetting, para a ação de subagrupamento. R tem muitos operadores poderosos para subsetting e dominá-los permitirá executar facilmente operações complexas em qualquer tipo de conjunto de dados. Existem seis maneiras diferentes de realizar essa operação em qualquer tipo de objeto, e três diferentes para as estruturas de dados. Vamos começar com os vetores atômicos:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
x
```

```
a   b   c   d   e
5.4 6.2 7.1 4.8 7.5
```

Então, agora que criamos um vetor fictício para brincar, como acessamos ao seu conteúdo?

Acessando elementos usando seus índices

Para extrair elementos de um vetor podemos informar seu índice correspondente, começando por um:

```
x[1]
```

```
a
5.4
```

```
x[4]
```

```
d
4.8
```

O operador de colchetes é como qualquer outra função. Para vetores atômicos (e matrizes), significa “me informe o elemento tal”.

Podemos pedir vários elementos ao mesmo tempo:

```
x[c(1, 3)]
```

```
a   c
5.4 7.1
```

Ou um “pedaço” do vetor:

```
x[1:4]
```

```
a   b   c   d
5.4 6.2 7.1 4.8
```

O operador : cria uma sequência de números do elemento esquerdo para o direito. Ou seja, `x[1: 4]`, é equivalente a `x[c(1,2,3,4)]`.

Podemos pedir o mesmo elemento várias vezes:

```
x[c(1,1,3)]
```

```
a   a   c
5.4 5.4 7.1
```

Se pedimos um número fora do vetor, R retornará valores ausentes:

```
x[6]
```

```
<NA>
```

```
NA
```

Este é um vetor de comprimento um, contendo um NA cujo nome é também NA.

Se pedimos o 0º elemento, obtemos um vetor vazio:

```
x[0]
```

```
named numeric(0)
```

Mas e os valores negativos?

Ignorando e remover elementos

Se usarmos um número negativo, R informará todos os elementos *exceto* o que foi especificado:

```
x[-2]
```

```
a c d e  
5.4 7.1 4.8 7.5
```

Podemos ignorar vários elementos:

```
x[c(-1, -5)] # ou x[-c(1,5)]
```

```
b c d  
6.2 7.1 4.8
```

Dica: Ordem de operações

Muitas pessoas tentam pular intervalos de um vetor. A maioria tenta primeiro negar uma sequência, dessa forma:

```
x[-1:3]
```

```
Error in x[-1:3]: only 0's may be mixed with negative subscripts
```

Lembre-se da ordem das operações! : É uma função, então o que acontece é que entende o seu primeiro argumento como -1, e o segundo como 3, gerando a sequência de números: c (-1, 0, 1, 2, 3).

A solução correta é “envolver” essa função entre parênteses. Assim o operador - será considerado nos resultados:

```
x[-(1:3)]
```

```
d e  
4.8 7.5
```

Para remover elementos de um vetor, precisamos atribuir os resultados de volta para a variável:

```
x <- x[-4]  
x
```

```
a b c e  
5.4 6.2 7.1 7.5
```

Desafio 1

Dado o seguinte código:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

1. apresente pelo menos 3 comandos diferentes que produzirão a seguinte saída:

```
{.r, echo=FALSE} x[2:4]
```

2. Compare as anotações com os demais. Vocês usaram estratégias diferentes?

Subset por nome

Podemos extrair elementos usando seu nome, em vez dos índices:

```
x[c("a", "c")]
```

```
a   c
5.4 7.1
```

Geralmente essa é uma maneira muito mais confiável de realizar subset de objetos: a posição de vários elementos pode muitas vezes mudar quando se encadeiam subsets, mas os nomes permanecerão sempre os mesmos! Infelizmente não podemos ignorar ou remover elementos tão facilmente.

Para ignorar (ou remover) um único elemento com nome:

```
x[-which(names(x) == "a")]
```

```
b   c   e
6.2 7.1 7.5
```

A função `which` retorna os índices de todos os elementos TRUE do seu argumento. Lembre-se de que as expressões são avaliadas antes de serem passadas para funções. Vamos por partes pra ficar claro como acontece.

Primeiro isso acontece:

```
names(x) == "a"
```

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

Esse operador condicionante é aplicado a cada nome do vetor `x`. Apenas o primeiro nome é `a` para que o elemento seja TRUE.

Agora `which` converte isso em um índice:

```
which(names(x) == "a")
```

```
[1] 1
```

Somente o primeiro elemento é TRUE, então `which` retorna 1. Agora que temos índices é possível ignorar outros dados porque temos um índice negativo!

Ignorar vários índices nomeados é semelhante, mas usa uma comparação diferente operador:

```
x[-which(names(x) %in% c("a", "c"))]
```

```
b   e
6.2 7.5
```

O `%in%` passa em cada elemento do seu argumento esquerdo, neste caso os nomes de `x`, e pergunta, “este elemento ocorre no segundo argumento?”.

Dica: Como obter ajuda para operadores

Podemos procurar ajuda nos operadores ao envolvê-los entre aspas: `Help ("%in%")` ou `?"%in%"`.

Subset através de outras operações lógicas

Podemos também realizar o subset de forma mais fácil através de operações lógicas:

```
a <- 1:10
b <- a > 7
a
[1] 1 2 3 4 5 6 7 8 9 10
b
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE
a[b]
[1] 8 9 10
a[a > 7]
[1] 8 9 10
```

Dica: Encadeando operações lógicas

Existem muitas situações em que você deseja combinar várias condicionantes. Para isso existem várias operações lógicas no R:

- `|` lógico OR : retorna TRUE, se à esquerda ou à direita forem TRUE
- `&` lógico AND: retorna TRUE se tanto o esquerdo, quanto direito forem TRUE
- `!` lógico NOT: converte TRUE paraFALSE e FALSE paraTRUE
- `&&` e `||` comparam os elementos individuais de dois vetores. Regras de reciclagem também se aplicam aqui.

Desafio

Dado o seguinte código:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
a b c d e
5.4 6.2 7.1 4.8 7.5
```

1. Escreva um comando subsetting para retornar os valores em `x` que são maiores que 4 e menores que 7.

Como lidar com valores especiais

Em algum momento encontramos funções no R que não podem lidar com dados faltantes, infinito ou indefinidos. Há uma série de funções especiais que podemos usar para filtrar esses dados:

- `is.na` retornará todas as posições em um vetor, matriz ou data.frame Contendo NA.
- Da mesma forma, `is.nan` e `is.infinite` farão o mesmo para NaN e Inf.
- `is.finite` retornará todas as posições em um vetor, matriz ou data.frame que não contenham «NA», «NaN» ou «Inf».
- `na.omit` irá omitir todos os valores faltantes (NA) de um vetor
- `is.na` retornará todas as locais em um vetor, matriz ou data.frame contendo «NA».
- Da mesma forma, `is.nan`, `is.infinite` farão o mesmo para NaN e Inf.
- `is.finite` retornará todas os locais em um vetor, matriz ou data.frame que não contenham «NA», «NaN» ou «Inf».
- `Na.omit` irá filtrar todos os valores faltantes de um vetor

Subset de fator

Agora que exploramos as diferentes formas de subset de vetores, como fazer o subset das outras estruturas de dados? O subset de fatores funciona da mesma maneira que o subset de vetores.

```
f <- factor(c("a", "a", "b", "c", "c", "d"))
f[f == "a"]
```

```
[1] a a
Levels: a b c d
f[f %in% c("b", "c")]
```

```
[1] b c c
Levels: a b c d
f[1:3]
```

```
[1] a a b
Levels: a b c d
```

Uma observação importante é que ignorar elementos não removerá o nível mesmo se não houver mais dessa categoria no fator:

```
f[-3]
```

```
[1] a a c c d
Levels: a b c d
```

Criando subset de Matriz

As matrizes também são manipuláveis usando a função `[`. Nesse caso usamos dois argumentos: o primeiro se aplicando a linhas, o segundo a colunas:

```
set.seed(1)
m <- matrix(rnorm(6*4), ncol=4, nrow=6)
m[3:4, c(3,1)]
```


	[,1]	[,2]	
[1,]	1.12493092	-0.8356286	
[2,]	-0.04493361	1.5952808	

O primeiro ou segundo argumento podem ser deixados em branco se quisermos usar todas as linhas ou colunas, respectivamente:

```
m[, c(3,4)]
```

```
[,1]      [,2]  
[1,] -0.62124058  0.82122120  
[2,] -2.21469989  0.59390132  
[3,]  1.12493092  0.91897737  
[4,] -0.04493361  0.78213630  
[5,] -0.01619026  0.07456498  
[6,]  0.94383621 -1.98935170
```

Se acessarmos apenas uma linha ou coluna, o R converterá automaticamente o resultado para um vector:

```
m[3,]
```

```
[1] -0.8356286  0.5757814  1.1249309  0.9189774
```

Se quisermos manter a saída como uma matriz, você precisa especificar um argumento *third; drop = FALSE*:

```
m[3, , drop=FALSE]
```

```
[,1]      [,2]      [,3]      [,4]  
[1,] -0.8356286  0.5757814  1.124931  0.9189774
```

Ao contrário dos vetores, se tentarmos acessar uma linha ou coluna fora da matriz, R irá acusar um erro:

```
m[, c(3,6)]
```

Tip: Arrays com muitas dimensões

Ao lidar com arrays multidimensionais, cada argumento para [Corresponde a uma dimensão. Por exemplo, numa matriz 3D, os três primeiros argumentos correspondem às linhas, colunas e dimensão de profundidade.

Como as matrizes são apenas vetores, podemos também fazer um subset usando apenas um argumento:

```
m[5]
```

```
[1] 0.3295078
```

Essa medida geralmente não é útil. No entanto, é bom notar que as matrizes são apresentadas no formato *column-major* por predefinição. Isso é, os elementos do vetor são organizados como *column-wise*:

```
matrix(1:6, nrow=2, ncol=3)
```

```
[,1] [,2] [,3]  
[1,]    1    3    5  
[2,]    2    4    6
```

Também podem ser gerados subset de matrizes usando seus nomes de linhas e de colunas em vez de seus índices de linha e coluna.

Desafio 2

Dado o seguinte código:

```
m <- matrix(1:18, nrow=3, ncol=6)
print(m)
```

```
[,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1     4     7    10    13    16
[2,]    2     5     8    11    14    17
[3,]    3     6     9    12    15    18
```

1. Qual dos seguintes comandos extrairá os valores 11 e 14?
- A. m[2,4,2,5]
 - B. m[2:5]
 - C. m[4:5,2]
 - D. m[2,c(4,5)]

Criando subset de lista

Existem três funções usadas para criar subsets de listas. `[`, Como vimos para vetores atômicos e matrizes, bem como `[[` e `$`. Usar `[` sempre nos retornará uma lista. Se quisermos fazer *subset* de uma lista, mas não *extrair* um elemento, então provavelmente usaremos `[`.

```
xlist <- list(a = "Software Carpentry", b = 1:10, data = head(iris))
xlist[1]
```

```
$a
[1] "Software Carpentry"
```

Isso retorna uma *lista com um elemento*.

Podemos fazer o subset dos elementos de uma lista exatamente como foi como vetores atômicos usando `[`. As operações de comparação, no entanto, não funcionará se os elementos não forem recursivos, eles tentarão condicionar as estruturas de dados em cada elemento da lista, não os elementos individuais dentro dessas estruturas de dados.

```
xlist[1:2]
```

```
$a
[1] "Software Carpentry"
```

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

Para extrair elementos individuais de uma lista, precisaremos usar a função de colchete duplo: `[[`.

```
xlist[[1]]
```

```
[1] "Software Carpentry"
```

Agora o resultado é um vetor, não uma lista.

Não podemos extrair mais de um elemento de uma só vez:

```
xlist[[1:2]]
```

```
Error in xlist[[1:2]]: subscript out of bounds
```

Nem usá-lo para ignorar elementos:

```
xlist[[-1]]  
  
Error in xlist[[-1]]: attempt to select more than one element
```

Mas podemos usar nomes para criar subsets e extrair elementos:

```
xlist[["a"]]  
  
[1] "Software Carpentry"
```

A função \$ é uma forma abreviada de extrair elementos pelo nome:

```
xlist$data  
  
Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
1           5.1         3.5          1.4         0.2   setosa  
2           4.9         3.0          1.4         0.2   setosa  
3           4.7         3.2          1.3         0.2   setosa  
4           4.6         3.1          1.5         0.2   setosa  
5           5.0         3.6          1.4         0.2   setosa  
6           5.4         3.9          1.7         0.4   setosa
```

Desafio 3

Dada a seguinte lista:

```
xlist <- list(a = "Software Carpentry", b = 1:10, data = head(iris))
```

Usando o seu conhecimento de criar subsets de lista e vetor, extraia o número 2 de xlist.

Dica: o número 2 está contido no item “b” na lista.

Desafio 4

Dado um modelo linear:

```
mod <- aov(pop ~ lifeExp, data=gapminder)
```

Extraia os graus residuais de liberdade (dica: tente usar `attributes()`)

Data frames

Lembre-se que os Data frames no fundo são listas, então são usadas regras similares. No entanto, eles também são objetos bidimensionais: [com um argumento atuará da mesma forma que para listas, onde cada lista corresponde a uma coluna. O objeto resultante será um data frame:

```
head(gapminder[3])
```

```
pop  
1 8425333  
2 9240934  
3 10267083  
4 11537966  
5 13079460  
6 14880372
```

Da mesma forma, [[será usado para extrair *uma única coluna*:

```
head(gapminder[["lifeExp"]])
```

```
[1] 28.801 30.332 31.997 34.020 36.088 38.438
```

E \$ fornece uma abreviatura funcional para extrair colunas pelo nome:

```
head(gapminder$year)
```

```
[1] 1952 1957 1962 1967 1972 1977
```

Com dois argumentos, [funciona da mesma maneira que para matrizes:

```
gapminder[1:3,]
```

	country	year	pop	continent	lifeExp	gdpPercap
1	Afghanistan	1952	8425333	Asia	28.801	779.4453
2	Afghanistan	1957	9240934	Asia	30.332	820.8530
3	Afghanistan	1962	10267083	Asia	31.997	853.1007

Se criamos um subset a partir de uma única linha, o resultado será um data frame (porque os elementos são tipos mistos):

```
gapminder[3,]
```

	country	year	pop	continent	lifeExp	gdpPercap
3	Afghanistan	1962	10267083	Asia	31.997	853.1007

Mas para uma única coluna o resultado será um vetor (isto pode ser alterado com o terceiro argumento, drop = FALSE).

Desafio 5

Corrija cada um dos seguintes erros de subset de data frame comuns:

1. Extraia as observações recolhidas para o ano 1957

```
gapminder[gapminder$year = 1957,]
```

2. Extraia todas as colunas, exceto de 1 a 4

```
gapminder[,-1:4]
```

3. Extraia as linhas onde a expectativa de vida é maior nos 80 anos

```
gapminder[gapminder$lifeExp > 80]
```

4. Extraia a primeira linha e a quarta e a quinta colunas (lifeExp e gdpPercap).

```
gapminder[1, 4, 5]
```

5. Avançando: extraia linhas que contenham informações para os anos de 2002 e 2007

```
gapminder[gapminder$year == 2002 | 2007,]
```

Desafio 6

1. Por que gapminder [1:20] retorna um erro? Como isso difere de gapminder[1:20,]?

2. Crie um novo `data.frame` chamado `gapminder_small` que contenha apenas as linhas de 1 a 9 E 19 a 23. Você pode fazer isso em uma ou duas etapas.

Soluções de desafio

Solução do desafio 1

Dado o seguinte código:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

a b c d e
5.4 6.2 7.1 4.8 7.5

1. Apresente pelo menos 3 comandos diferentes que produzirão o seguinte resultado:

b c d
6.2 7.1 4.8

```
x[2:4]
x[-c(1,5)]
x[c("b", "c", "d")]
x[c(2,3,4)]
```

Solução do desafio 2

Dado o seguinte código:

```
m <- matrix(1:18, nrow=3, ncol=6)
print(m)
```

[,1]	[,2]	[,3]	[,4]	[,5]	[,6]
[1,]	1	4	7	10	13
[2,]	2	5	8	11	14
[3,]	3	6	9	12	15

1. Qual dos seguintes comandos extrairá os valores 11 e 14?

- A. `m[2,4,2,5]`
- B. `m[2:5]`
- C. `m[4:5,2]`
- D. `m[2,c(4,5)]`

Answer: D

Solução do desafio 3

Dada a seguinte lista:

```
xlist <- list(a = "Software Carpentry", b = 1:10, data = head(iris))
```

A partir das formas de criar subsets de lista e vetor, extraia o número 2 de xlist. Dica: o número 2 está contido no item “b” na lista.

```
xlist$b[2]  
xlist[[2]][2]  
xlist[["b"]][2]
```

Solução do desafio 4

Dado um modelo linear:

```
mod <- aov(pop ~ lifeExp, data=gapminder)
```

Extraia os graus residuais de liberdade (dica: `attributes()` ajudará)

```
attributes(mod) ## `df.residual` é um dos nomes de `mod`  
mod$df.residual
```

Solução do desafio 5

Corrija cada um dos seguintes erros comuns de criar subset de data frame:

1. Extrair as observações recolhidas para o ano 1957

```
# gapminder[gapminder$year = 1957,]  
gapminder[gapminder$year == 1957,]
```

2. Extrair todas as colunas, exceto de 1 a 4

```
# gapminder[,-1:4]  
gapminder[,-c(1:4)]
```

3. Extraia as linhas onde a expectativa de vida é maior nos 80 anos

```
# gapminder[gapminder$lifeExp > 80]  
gapminder[gapminder$lifeExp > 80,]
```

4. Extraia a primeira linha e a quarta e a quinta colunas (`lifeExp` and `gdpPercap`).

```
# gapminder[1, 4, 5]  
gapminder[1, c(4, 5)]
```

5. Avançado: extraia linhas que contêm informações para os anos de 2002 E 2007

```
# gapminder[gapminder$year == 2002 | 2007,]  
gapminder[gapminder$year == 2002 | gapminder$year == 2007,]  
gapminder[gapminder$year %in% c(2002, 2007),]
```

Solução do desafio 6

1. Por que `gapminder [1:20]` retorna um erro? Como isso difere de `gapminder [1:20,]`?

Resposta: `gapminder` é um `data.frame` e por isso precisa ser criado um subset em duas dimensões. Fazer o subset dos dados de `Gapminder [1:20,]` para selecionar as primeiras 20 linhas e todas as colunas.

2. Crie um novo `data.frame` chamado `gapminder_small` que contenha apenas as linhas de 1 a 9 e de 19 a 23. Você pode fazer isso em uma ou duas etapas.

```
gapminder_small <- gapminder[c(1:9, 19:23),]
```

Lendo dados

Mais acima obtivemos os dados chamados `gapminder`. Curioso de onde vêm esses dados? Dê uma olhada no site da Gapminder Agora vamos carregar os dados `gapminder` em R. Como a extensão (.csv) do arquivo sugere, ele contém valores separados por vírgula, e parece conter uma linha de cabeçalho. Podemos usar `read.table` para ler o arquivo no R. O comando `read.table` lê um arquivo como um `data frame`.

```
gapminder <- read.table(file = "/home/saulo/Documents/projetos/Cursos e oficinas/workshop R/r-novice-gapminder.csv", header = TRUE, sep = ",")
```

Como já saibamos a estrutura dos dados, podemos especificar os argumentos apropriados para `read.table`. Sem estes argumentos, `read.table` fará o seu melhor para ler o arquivo de forma sensata, mas é sempre mais confiável dizer explicitamente ao `read.table` a estrutura dos dados. A função `read.csv` fornece um atalho apropriado para carregar arquivos CSV.

Dicas Diversas

1. Outros tipos de arquivos que podemos encontrar são os formatos separados por TAB. Para especificar uma guia como um separador, use "\t".
2. É possível ler arquivos na Internet, substituindo os caminhos de arquivo por um endereço da Web.
3. Também é possível ler diretamente de planilhas Excel sem convertê-las previamente para texto simples usando o pacote `xlsx`.

Desafio 2

Ir para file -> new file -> R script, e escreva um script R para carregar no conjunto de dados `gapminder`. Coloque-o no diretório `scripts/` e adicione-o ao controle de versão.

Execute o script usando a função `source`, utilizando o caminho do arquivo como seu argumento (ou pressionando o botão “source” no RStudio).

Usando data frames: o conjunto de dados `gapminder`

Para recapitular vamos olhar nosso exemplo dados (expectativa de vida em vários países durante vários anos). Importante lembrar que existem algumas funções que podemos usar para “questionar” as estruturas de dados em R:

```

class() # qual é a estrutura de dados?
typeof() # qual é o seu tipo atômico?
length() # qual o tamanho? E sobre objetos bidimensionais?
attributes() # tem algum metadado?
str() # resumo completo de todo o objeto
dim() # Dimensões do objeto - também tente nrow (), ncol ()

```

Explorando o conjunto de dados gapminder.

```
typeof(gapminder)
```

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

Lembrando que data frames são listas “encobertas”.

```
class(gapminder)
```

```
[1] "data.frame"
```

Os dados gapminder são armazenados em um “data.frame”. Esta é a estrutura de dados padrão quando lemos dados, e é útil para armazenar dados com tipos mistos de colunas.

Desafio 3: Tipos de dados em um conjunto de dados real

Olhe para as primeiras 6 linhas do data frame do gapminder que foi carregado anteriormente:

```
head(gapminder)
```

	country	year	pop	continent	lifeExp	gdpPercap
1	Afghanistan	1952	8425333	Asia	28.801	779.4453
2	Afghanistan	1957	9240934	Asia	30.332	820.8530
3	Afghanistan	1962	10267083	Asia	31.997	853.1007
4	Afghanistan	1967	11537966	Asia	34.020	836.1971
5	Afghanistan	1972	13079460	Asia	36.088	739.9811
6	Afghanistan	1977	14880372	Asia	38.438	786.1134

Anote o tipo de dados que você acha que está em cada coluna

```
typeof(gapminder$year)
```

```
[1] "integer"
```

```
typeof(gapminder$lifeExp)
```

```
[1] "double"
```

Descobrindo de qual tipo será a coluna de continente?

```
typeof(gapminder$continent)
```

```
[1] "integer"
```

Explorando a classe desta coluna. A resposta esperada poderia ser “caractere”?

```
class(gapminder$continent)
```

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

Uma das ações padrão de R é tratar todas as colunas de texto como “fatores” quando lê dados. A razão para isso é que as colunas de texto frequentemente representam dados categóricos, que precisam ser fatores a

serem tratados adequadamente por funções de modelagem estatística no R. No entanto, não é um padrão óbvio, e algo que muitas pessoas se confundem. Nós podemos desativar esse padrão e ler os dados novamente. Lembre-se, se a conversão automática para fatores for desativada, será preciso explicitar a necessidade de conversão das variáveis em fatores quando for rodar modelos estatísticos. Isso pode ser útil, porque nos força a pensarmos nas perguntas que estamos fazendo, e torna mais fácil para especificar a ordenação das categorias.

Dica: Alterando opções

Quando R inicia, a primeira coisa que faz é executar qualquer código no arquivo `.Rprofile` no diretório do projeto. Quaisquer alterações permanentes na configuração padrão que for feita devem ser armazenados nesse arquivo.

A primeira coisa que deve ser feita ao ler dados é verificar se ele corresponde ao desejado, mesmo se o comando tenha executado sem avisos ou erros. A função `str`, abreviação de “estrutura”, é realmente útil para isso:

```
str(gapminder)
```

```
'data.frame': 1704 obs. of 6 variables:
 $ country : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 ...
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...
```

Podemos ver que o objeto é um `data.frame` com 1.704 observações (linhas), e 6 variáveis (colunas). Abaixo disso, vemos o nome de cada coluna, seguido Por um “`:`”, seguido pelo tipo de variável nessa coluna, juntamente com as primeiras poucas entradas.

Nós também podemos recuperar ou modificar a coluna ou rownames do `data.frame`:

```
colnames(gapminder)
```

```
[1] "country"   "year"       "pop"        "continent"  "lifeExp"    "gdpPercap"
```

```
rownames(gapminder)
```

```
[1] "1"      "2"      "3"      "4"      "5"      "6"      "7"      "8"      "9"
[10] "10"     "11"     "12"     "13"     "14"     "15"     "16"     "17"     "18"
[19] "19"     "20"     "21"     "22"     "23"     "24"     "25"     "26"     "27"
[28] "28"     "29"     "30"     "31"     "32"     "33"     "34"     "35"     "36"
[37] "37"     "38"     "39"     "40"     "41"     "42"     "43"     "44"     "45"
[46] "46"     "47"     "48"     "49"     "50"     "51"     "52"     "53"     "54"
[55] "55"     "56"     "57"     "58"     "59"     "60"     "61"     "62"     "63"
[64] "64"     "65"     "66"     "67"     "68"     "69"     "70"     "71"     "72"
[73] "73"     "74"     "75"     "76"     "77"     "78"     "79"     "80"     "81"
[82] "82"     "83"     "84"     "85"     "86"     "87"     "88"     "89"     "90"
[91] "91"     "92"     "93"     "94"     "95"     "96"     "97"     "98"     "99"
[100] "100"    "101"    "102"    "103"    "104"    "105"    "106"    "107"    "108"
[109] "109"    "110"    "111"    "112"    "113"    "114"    "115"    "116"    "117"
[118] "118"    "119"    "120"    "121"    "122"    "123"    "124"    "125"    "126"
[127] "127"    "128"    "129"    "130"    "131"    "132"    "133"    "134"    "135"
[136] "136"    "137"    "138"    "139"    "140"    "141"    "142"    "143"    "144"
[145] "145"    "146"    "147"    "148"    "149"    "150"    "151"    "152"    "153"
[154] "154"    "155"    "156"    "157"    "158"    "159"    "160"    "161"    "162"
```

[163]	"163"	"164"	"165"	"166"	"167"	"168"	"169"	"170"	"171"
[172]	"172"	"173"	"174"	"175"	"176"	"177"	"178"	"179"	"180"
[181]	"181"	"182"	"183"	"184"	"185"	"186"	"187"	"188"	"189"
[190]	"190"	"191"	"192"	"193"	"194"	"195"	"196"	"197"	"198"
[199]	"199"	"200"	"201"	"202"	"203"	"204"	"205"	"206"	"207"
[208]	"208"	"209"	"210"	"211"	"212"	"213"	"214"	"215"	"216"
[217]	"217"	"218"	"219"	"220"	"221"	"222"	"223"	"224"	"225"
[226]	"226"	"227"	"228"	"229"	"230"	"231"	"232"	"233"	"234"
[235]	"235"	"236"	"237"	"238"	"239"	"240"	"241"	"242"	"243"
[244]	"244"	"245"	"246"	"247"	"248"	"249"	"250"	"251"	"252"
[253]	"253"	"254"	"255"	"256"	"257"	"258"	"259"	"260"	"261"
[262]	"262"	"263"	"264"	"265"	"266"	"267"	"268"	"269"	"270"
[271]	"271"	"272"	"273"	"274"	"275"	"276"	"277"	"278"	"279"
[280]	"280"	"281"	"282"	"283"	"284"	"285"	"286"	"287"	"288"
[289]	"289"	"290"	"291"	"292"	"293"	"294"	"295"	"296"	"297"
[298]	"298"	"299"	"300"	"301"	"302"	"303"	"304"	"305"	"306"
[307]	"307"	"308"	"309"	"310"	"311"	"312"	"313"	"314"	"315"
[316]	"316"	"317"	"318"	"319"	"320"	"321"	"322"	"323"	"324"
[325]	"325"	"326"	"327"	"328"	"329"	"330"	"331"	"332"	"333"
[334]	"334"	"335"	"336"	"337"	"338"	"339"	"340"	"341"	"342"
[343]	"343"	"344"	"345"	"346"	"347"	"348"	"349"	"350"	"351"
[352]	"352"	"353"	"354"	"355"	"356"	"357"	"358"	"359"	"360"
[361]	"361"	"362"	"363"	"364"	"365"	"366"	"367"	"368"	"369"
[370]	"370"	"371"	"372"	"373"	"374"	"375"	"376"	"377"	"378"
[379]	"379"	"380"	"381"	"382"	"383"	"384"	"385"	"386"	"387"
[388]	"388"	"389"	"390"	"391"	"392"	"393"	"394"	"395"	"396"
[397]	"397"	"398"	"399"	"400"	"401"	"402"	"403"	"404"	"405"
[406]	"406"	"407"	"408"	"409"	"410"	"411"	"412"	"413"	"414"
[415]	"415"	"416"	"417"	"418"	"419"	"420"	"421"	"422"	"423"
[424]	"424"	"425"	"426"	"427"	"428"	"429"	"430"	"431"	"432"
[433]	"433"	"434"	"435"	"436"	"437"	"438"	"439"	"440"	"441"
[442]	"442"	"443"	"444"	"445"	"446"	"447"	"448"	"449"	"450"
[451]	"451"	"452"	"453"	"454"	"455"	"456"	"457"	"458"	"459"
[460]	"460"	"461"	"462"	"463"	"464"	"465"	"466"	"467"	"468"
[469]	"469"	"470"	"471"	"472"	"473"	"474"	"475"	"476"	"477"
[478]	"478"	"479"	"480"	"481"	"482"	"483"	"484"	"485"	"486"
[487]	"487"	"488"	"489"	"490"	"491"	"492"	"493"	"494"	"495"
[496]	"496"	"497"	"498"	"499"	"500"	"501"	"502"	"503"	"504"
[505]	"505"	"506"	"507"	"508"	"509"	"510"	"511"	"512"	"513"
[514]	"514"	"515"	"516"	"517"	"518"	"519"	"520"	"521"	"522"
[523]	"523"	"524"	"525"	"526"	"527"	"528"	"529"	"530"	"531"
[532]	"532"	"533"	"534"	"535"	"536"	"537"	"538"	"539"	"540"
[541]	"541"	"542"	"543"	"544"	"545"	"546"	"547"	"548"	"549"
[550]	"550"	"551"	"552"	"553"	"554"	"555"	"556"	"557"	"558"
[559]	"559"	"560"	"561"	"562"	"563"	"564"	"565"	"566"	"567"
[568]	"568"	"569"	"570"	"571"	"572"	"573"	"574"	"575"	"576"
[577]	"577"	"578"	"579"	"580"	"581"	"582"	"583"	"584"	"585"
[586]	"586"	"587"	"588"	"589"	"590"	"591"	"592"	"593"	"594"
[595]	"595"	"596"	"597"	"598"	"599"	"600"	"601"	"602"	"603"
[604]	"604"	"605"	"606"	"607"	"608"	"609"	"610"	"611"	"612"
[613]	"613"	"614"	"615"	"616"	"617"	"618"	"619"	"620"	"621"
[622]	"622"	"623"	"624"	"625"	"626"	"627"	"628"	"629"	"630"
[631]	"631"	"632"	"633"	"634"	"635"	"636"	"637"	"638"	"639"
[640]	"640"	"641"	"642"	"643"	"644"	"645"	"646"	"647"	"648"

[649]	"649"	"650"	"651"	"652"	"653"	"654"	"655"	"656"	"657"
[658]	"658"	"659"	"660"	"661"	"662"	"663"	"664"	"665"	"666"
[667]	"667"	"668"	"669"	"670"	"671"	"672"	"673"	"674"	"675"
[676]	"676"	"677"	"678"	"679"	"680"	"681"	"682"	"683"	"684"
[685]	"685"	"686"	"687"	"688"	"689"	"690"	"691"	"692"	"693"
[694]	"694"	"695"	"696"	"697"	"698"	"699"	"700"	"701"	"702"
[703]	"703"	"704"	"705"	"706"	"707"	"708"	"709"	"710"	"711"
[712]	"712"	"713"	"714"	"715"	"716"	"717"	"718"	"719"	"720"
[721]	"721"	"722"	"723"	"724"	"725"	"726"	"727"	"728"	"729"
[730]	"730"	"731"	"732"	"733"	"734"	"735"	"736"	"737"	"738"
[739]	"739"	"740"	"741"	"742"	"743"	"744"	"745"	"746"	"747"
[748]	"748"	"749"	"750"	"751"	"752"	"753"	"754"	"755"	"756"
[757]	"757"	"758"	"759"	"760"	"761"	"762"	"763"	"764"	"765"
[766]	"766"	"767"	"768"	"769"	"770"	"771"	"772"	"773"	"774"
[775]	"775"	"776"	"777"	"778"	"779"	"780"	"781"	"782"	"783"
[784]	"784"	"785"	"786"	"787"	"788"	"789"	"790"	"791"	"792"
[793]	"793"	"794"	"795"	"796"	"797"	"798"	"799"	"800"	"801"
[802]	"802"	"803"	"804"	"805"	"806"	"807"	"808"	"809"	"810"
[811]	"811"	"812"	"813"	"814"	"815"	"816"	"817"	"818"	"819"
[820]	"820"	"821"	"822"	"823"	"824"	"825"	"826"	"827"	"828"
[829]	"829"	"830"	"831"	"832"	"833"	"834"	"835"	"836"	"837"
[838]	"838"	"839"	"840"	"841"	"842"	"843"	"844"	"845"	"846"
[847]	"847"	"848"	"849"	"850"	"851"	"852"	"853"	"854"	"855"
[856]	"856"	"857"	"858"	"859"	"860"	"861"	"862"	"863"	"864"
[865]	"865"	"866"	"867"	"868"	"869"	"870"	"871"	"872"	"873"
[874]	"874"	"875"	"876"	"877"	"878"	"879"	"880"	"881"	"882"
[883]	"883"	"884"	"885"	"886"	"887"	"888"	"889"	"890"	"891"
[892]	"892"	"893"	"894"	"895"	"896"	"897"	"898"	"899"	"900"
[901]	"901"	"902"	"903"	"904"	"905"	"906"	"907"	"908"	"909"
[910]	"910"	"911"	"912"	"913"	"914"	"915"	"916"	"917"	"918"
[919]	"919"	"920"	"921"	"922"	"923"	"924"	"925"	"926"	"927"
[928]	"928"	"929"	"930"	"931"	"932"	"933"	"934"	"935"	"936"
[937]	"937"	"938"	"939"	"940"	"941"	"942"	"943"	"944"	"945"
[946]	"946"	"947"	"948"	"949"	"950"	"951"	"952"	"953"	"954"
[955]	"955"	"956"	"957"	"958"	"959"	"960"	"961"	"962"	"963"
[964]	"964"	"965"	"966"	"967"	"968"	"969"	"970"	"971"	"972"
[973]	"973"	"974"	"975"	"976"	"977"	"978"	"979"	"980"	"981"
[982]	"982"	"983"	"984"	"985"	"986"	"987"	"988"	"989"	"990"
[991]	"991"	"992"	"993"	"994"	"995"	"996"	"997"	"998"	"999"
[1000]	"1000"	"1001"	"1002"	"1003"	"1004"	"1005"	"1006"	"1007"	"1008"
[1009]	"1009"	"1010"	"1011"	"1012"	"1013"	"1014"	"1015"	"1016"	"1017"
[1018]	"1018"	"1019"	"1020"	"1021"	"1022"	"1023"	"1024"	"1025"	"1026"
[1027]	"1027"	"1028"	"1029"	"1030"	"1031"	"1032"	"1033"	"1034"	"1035"
[1036]	"1036"	"1037"	"1038"	"1039"	"1040"	"1041"	"1042"	"1043"	"1044"
[1045]	"1045"	"1046"	"1047"	"1048"	"1049"	"1050"	"1051"	"1052"	"1053"
[1054]	"1054"	"1055"	"1056"	"1057"	"1058"	"1059"	"1060"	"1061"	"1062"
[1063]	"1063"	"1064"	"1065"	"1066"	"1067"	"1068"	"1069"	"1070"	"1071"
[1072]	"1072"	"1073"	"1074"	"1075"	"1076"	"1077"	"1078"	"1079"	"1080"
[1081]	"1081"	"1082"	"1083"	"1084"	"1085"	"1086"	"1087"	"1088"	"1089"
[1090]	"1090"	"1091"	"1092"	"1093"	"1094"	"1095"	"1096"	"1097"	"1098"
[1099]	"1099"	"1100"	"1101"	"1102"	"1103"	"1104"	"1105"	"1106"	"1107"
[1108]	"1108"	"1109"	"1110"	"1111"	"1112"	"1113"	"1114"	"1115"	"1116"
[1117]	"1117"	"1118"	"1119"	"1120"	"1121"	"1122"	"1123"	"1124"	"1125"
[1126]	"1126"	"1127"	"1128"	"1129"	"1130"	"1131"	"1132"	"1133"	"1134"

[1135] "1135" "1136" "1137" "1138" "1139" "1140" "1141" "1142" "1143"
 [1144] "1144" "1145" "1146" "1147" "1148" "1149" "1150" "1151" "1152"
 [1153] "1153" "1154" "1155" "1156" "1157" "1158" "1159" "1160" "1161"
 [1162] "1162" "1163" "1164" "1165" "1166" "1167" "1168" "1169" "1170"
 [1171] "1171" "1172" "1173" "1174" "1175" "1176" "1177" "1178" "1179"
 [1180] "1180" "1181" "1182" "1183" "1184" "1185" "1186" "1187" "1188"
 [1189] "1189" "1190" "1191" "1192" "1193" "1194" "1195" "1196" "1197"
 [1198] "1198" "1199" "1200" "1201" "1202" "1203" "1204" "1205" "1206"
 [1207] "1207" "1208" "1209" "1210" "1211" "1212" "1213" "1214" "1215"
 [1216] "1216" "1217" "1218" "1219" "1220" "1221" "1222" "1223" "1224"
 [1225] "1225" "1226" "1227" "1228" "1229" "1230" "1231" "1232" "1233"
 [1234] "1234" "1235" "1236" "1237" "1238" "1239" "1240" "1241" "1242"
 [1243] "1243" "1244" "1245" "1246" "1247" "1248" "1249" "1250" "1251"
 [1252] "1252" "1253" "1254" "1255" "1256" "1257" "1258" "1259" "1260"
 [1261] "1261" "1262" "1263" "1264" "1265" "1266" "1267" "1268" "1269"
 [1270] "1270" "1271" "1272" "1273" "1274" "1275" "1276" "1277" "1278"
 [1279] "1279" "1280" "1281" "1282" "1283" "1284" "1285" "1286" "1287"
 [1288] "1288" "1289" "1290" "1291" "1292" "1293" "1294" "1295" "1296"
 [1297] "1297" "1298" "1299" "1300" "1301" "1302" "1303" "1304" "1305"
 [1306] "1306" "1307" "1308" "1309" "1310" "1311" "1312" "1313" "1314"
 [1315] "1315" "1316" "1317" "1318" "1319" "1320" "1321" "1322" "1323"
 [1324] "1324" "1325" "1326" "1327" "1328" "1329" "1330" "1331" "1332"
 [1333] "1333" "1334" "1335" "1336" "1337" "1338" "1339" "1340" "1341"
 [1342] "1342" "1343" "1344" "1345" "1346" "1347" "1348" "1349" "1350"
 [1351] "1351" "1352" "1353" "1354" "1355" "1356" "1357" "1358" "1359"
 [1360] "1360" "1361" "1362" "1363" "1364" "1365" "1366" "1367" "1368"
 [1369] "1369" "1370" "1371" "1372" "1373" "1374" "1375" "1376" "1377"
 [1378] "1378" "1379" "1380" "1381" "1382" "1383" "1384" "1385" "1386"
 [1387] "1387" "1388" "1389" "1390" "1391" "1392" "1393" "1394" "1395"
 [1396] "1396" "1397" "1398" "1399" "1400" "1401" "1402" "1403" "1404"
 [1405] "1405" "1406" "1407" "1408" "1409" "1410" "1411" "1412" "1413"
 [1414] "1414" "1415" "1416" "1417" "1418" "1419" "1420" "1421" "1422"
 [1423] "1423" "1424" "1425" "1426" "1427" "1428" "1429" "1430" "1431"
 [1432] "1432" "1433" "1434" "1435" "1436" "1437" "1438" "1439" "1440"
 [1441] "1441" "1442" "1443" "1444" "1445" "1446" "1447" "1448" "1449"
 [1450] "1450" "1451" "1452" "1453" "1454" "1455" "1456" "1457" "1458"
 [1459] "1459" "1460" "1461" "1462" "1463" "1464" "1465" "1466" "1467"
 [1468] "1468" "1469" "1470" "1471" "1472" "1473" "1474" "1475" "1476"
 [1477] "1477" "1478" "1479" "1480" "1481" "1482" "1483" "1484" "1485"
 [1486] "1486" "1487" "1488" "1489" "1490" "1491" "1492" "1493" "1494"
 [1495] "1495" "1496" "1497" "1498" "1499" "1500" "1501" "1502" "1503"
 [1504] "1504" "1505" "1506" "1507" "1508" "1509" "1510" "1511" "1512"
 [1513] "1513" "1514" "1515" "1516" "1517" "1518" "1519" "1520" "1521"
 [1522] "1522" "1523" "1524" "1525" "1526" "1527" "1528" "1529" "1530"
 [1531] "1531" "1532" "1533" "1534" "1535" "1536" "1537" "1538" "1539"
 [1540] "1540" "1541" "1542" "1543" "1544" "1545" "1546" "1547" "1548"
 [1549] "1549" "1550" "1551" "1552" "1553" "1554" "1555" "1556" "1557"
 [1558] "1558" "1559" "1560" "1561" "1562" "1563" "1564" "1565" "1566"
 [1567] "1567" "1568" "1569" "1570" "1571" "1572" "1573" "1574" "1575"
 [1576] "1576" "1577" "1578" "1579" "1580" "1581" "1582" "1583" "1584"
 [1585] "1585" "1586" "1587" "1588" "1589" "1590" "1591" "1592" "1593"
 [1594] "1594" "1595" "1596" "1597" "1598" "1599" "1600" "1601" "1602"
 [1603] "1603" "1604" "1605" "1606" "1607" "1608" "1609" "1610" "1611"
 [1612] "1612" "1613" "1614" "1615" "1616" "1617" "1618" "1619" "1620"

```
[1621] "1621" "1622" "1623" "1624" "1625" "1626" "1627" "1628" "1629"
[1630] "1630" "1631" "1632" "1633" "1634" "1635" "1636" "1637" "1638"
[1639] "1639" "1640" "1641" "1642" "1643" "1644" "1645" "1646" "1647"
[1648] "1648" "1649" "1650" "1651" "1652" "1653" "1654" "1655" "1656"
[1657] "1657" "1658" "1659" "1660" "1661" "1662" "1663" "1664" "1665"
[1666] "1666" "1667" "1668" "1669" "1670" "1671" "1672" "1673" "1674"
[1675] "1675" "1676" "1677" "1678" "1679" "1680" "1681" "1682" "1683"
[1684] "1684" "1685" "1686" "1687" "1688" "1689" "1690" "1691" "1692"
[1693] "1693" "1694" "1695" "1696" "1697" "1698" "1699" "1700" "1701"
[1702] "1702" "1703" "1704"
```

Os números nos colchetes à esquerda informam o número da primeira entrada na linha do resultado. Assim, na última linha vemos que o elemento “[1701]” tem “1701” armazenado nele. Os “rownames” neste caso são simplesmente os números de linha.

Podemos também modificar esta informação:

```
copy <- gapminder # permite criar uma cópia para não detonar o original
colnames(copy) <- c("a", "b", "c", "d", "e", "f")
head(copy)
```

	a	b	c	d	e	f
1	Afghanistan	1952	8425333	Asia	28.801	779.4453
2	Afghanistan	1957	9240934	Asia	30.332	820.8530
3	Afghanistan	1962	10267083	Asia	31.997	853.1007
4	Afghanistan	1967	11537966	Asia	34.020	836.1971
5	Afghanistan	1972	13079460	Asia	36.088	739.9811
6	Afghanistan	1977	14880372	Asia	38.438	786.1134

Existem algumas maneiras de recuperar e modificar essas informações. **attributes** fornece os nomes de linha e coluna, junto às informações de classe, enquanto **dimnames** informa apenas os rownames e nomes de colunas.

Em ambos os casos, o objeto resultante é armazenado em uma lista:

```
str(dimnames(gapminder))
```

```
List of 2
$ : chr [1:1704] "1" "2" "3" "4" ...
$ : chr [1:6] "country" "year" "pop" "continent" ...
```

Entendendo como as listas são usadas nos “output” de função.

Vamos executar uma regressão linear básica no conjunto de dados gapminder:

```
# Qual é a relação entre a expectativa de vida eo ano?
11 <- lm(lifeExp ~ year, data=gapminder)
```

Brevemente: O ~ denota uma fórmula, o que significa tratar a variável à esquerda do ~ como a lado esquerdo da equação (ou variável resposta, neste caso), e tudo à direita como o lado direito. Informando à função do modelo para usar data frame do gapminder, ela sabe onde buscar pelas variáveis e suas colunas.

Vejamos a saída:

```
11
```

```
Call:
lm(formula = lifeExp ~ year, data = gapminder)
```

Coefficients:

```
(Intercept)      year  
-585.6522     0.3259
```

Nada demais, certo? Mas se olharmos para a estrutura...

```
str(l1)
```

List of 12

```
$ coefficients : Named num [1:2] -585.652 0.326  
..- attr(*, "names")= chr [1:2] "(Intercept)" "year"  
$ residuals    : Named num [1:1704] -21.7 -21.8 -21.8 -21.4 -20.9 ...  
..- attr(*, "names")= chr [1:1704] "1" "2" "3" "4" ...  
$ effects      : Named num [1:1704] -2455.1 232.2 -20.8 -20.5 -20.2 ...  
..- attr(*, "names")= chr [1:1704] "(Intercept)" "year" "" "" ...  
$ rank         : int 2  
$ fitted.values: Named num [1:1704] 50.5 52.1 53.8 55.4 57 ...  
..- attr(*, "names")= chr [1:1704] "1" "2" "3" "4" ...  
$ assign        : int [1:2] 0 1  
$ qr            :List of 5  
..$ qr   : num [1:1704, 1:2] -41.2795 0.0242 0.0242 0.0242 0.0242 ...  
... ..- attr(*, "dimnames")=List of 2  
... ... .$: chr [1:1704] "1" "2" "3" "4" ...  
... ... .$: chr [1:2] "(Intercept)" "year"  
... ..- attr(*, "assign")= int [1:2] 0 1  
..$ qraux: num [1:2] 1.02 1.03  
..$ pivot: int [1:2] 1 2  
..$ tol  : num 1e-07  
..$ rank  : int 2  
..- attr(*, "class")= chr "qr"  
$ df.residual  : int 1702  
$ xlevels      : Named list()  
$ call          : language lm(formula = lifeExp ~ year, data = gapminder)  
$ terms         :Classes 'terms', 'formula' length 3 lifeExp ~ year  
... ..- attr(*, "variables")= language list(lifeExp, year)  
... ..- attr(*, "factors")= int [1:2, 1] 0 1  
... ..- attr(*, "dimnames")=List of 2  
... ... .$: chr [1:2] "lifeExp" "year"  
... ... .$: chr "year"  
... ..- attr(*, "term.labels")= chr "year"  
... ..- attr(*, "order")= int 1  
... ..- attr(*, "intercept")= int 1  
... ..- attr(*, "response")= int 1  
... ..- attr(*, ".Environment")=<environment: R_GlobalEnv>  
... ..- attr(*, "predvars")= language list(lifeExp, year)  
... ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"  
... ... ..- attr(*, "names")= chr [1:2] "lifeExp" "year"  
$ model         :'data.frame': 1704 obs. of 2 variables:  
..$ lifeExp: num [1:1704] 28.8 30.3 32 34 36.1 ...  
..$ year   : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...  
..- attr(*, "terms")=Classes 'terms', 'formula' length 3 lifeExp ~ year  
... ..- attr(*, "variables")= language list(lifeExp, year)  
... ..- attr(*, "factors")= int [1:2, 1] 0 1  
... ..- attr(*, "dimnames")=List of 2
```

```

... . . . . . $ : chr [1:2] "lifeExp" "year"
... . . . . . $ : chr "year"
... . . . - attr(*, "term.labels")= chr "year"
... . . . - attr(*, "order")= int 1
... . . . - attr(*, "intercept")= int 1
... . . . - attr(*, "response")= int 1
... . . . - attr(*, ".Environment")=<environment: R_GlobalEnv>
... . . . - attr(*, "predvars")= language list(lifeExp, year)
... . . . - attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
... . . . - attr(*, "names")= chr [1:2] "lifeExp" "year"
- attr(*, "class")= chr "lm"

```

Há várias coisas armazenadas em listas, de forma aninhadas! É por isso que a função estrutura é muito útil, permitindo ver todos os dados disponíveis.

Agora podemos olhar para o `summary`:

```
summary(l1)
```

Call:
`lm(formula = lifeExp ~ year, data = gapminder)`

Residuals:

Min	1Q	Median	3Q	Max
-39.949	-9.651	1.697	10.335	22.158

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-585.65219	32.31396	-18.12	<2e-16 ***
year	0.32590	0.01632	19.96	<2e-16 ***

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

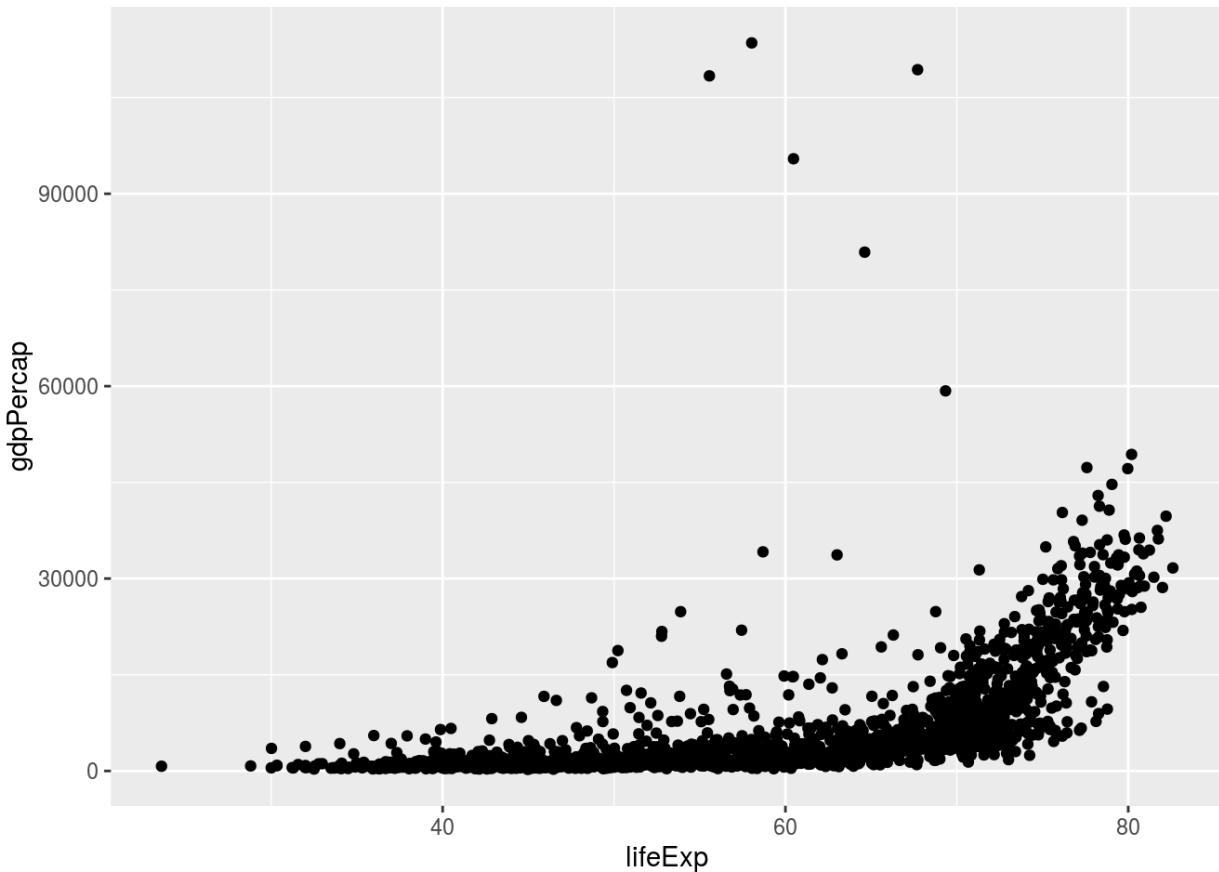
Residual standard error: 11.63 on 1702 degrees of freedom
Multiple R-squared: 0.1898, Adjusted R-squared: 0.1893
F-statistic: 398.6 on 1 and 1702 DF, p-value: < 2.2e-16

Como esperado, a expectativa de vida tem aumentado lentamente ao longo do tempo, então vemos uma associação positiva significativa!

Plots

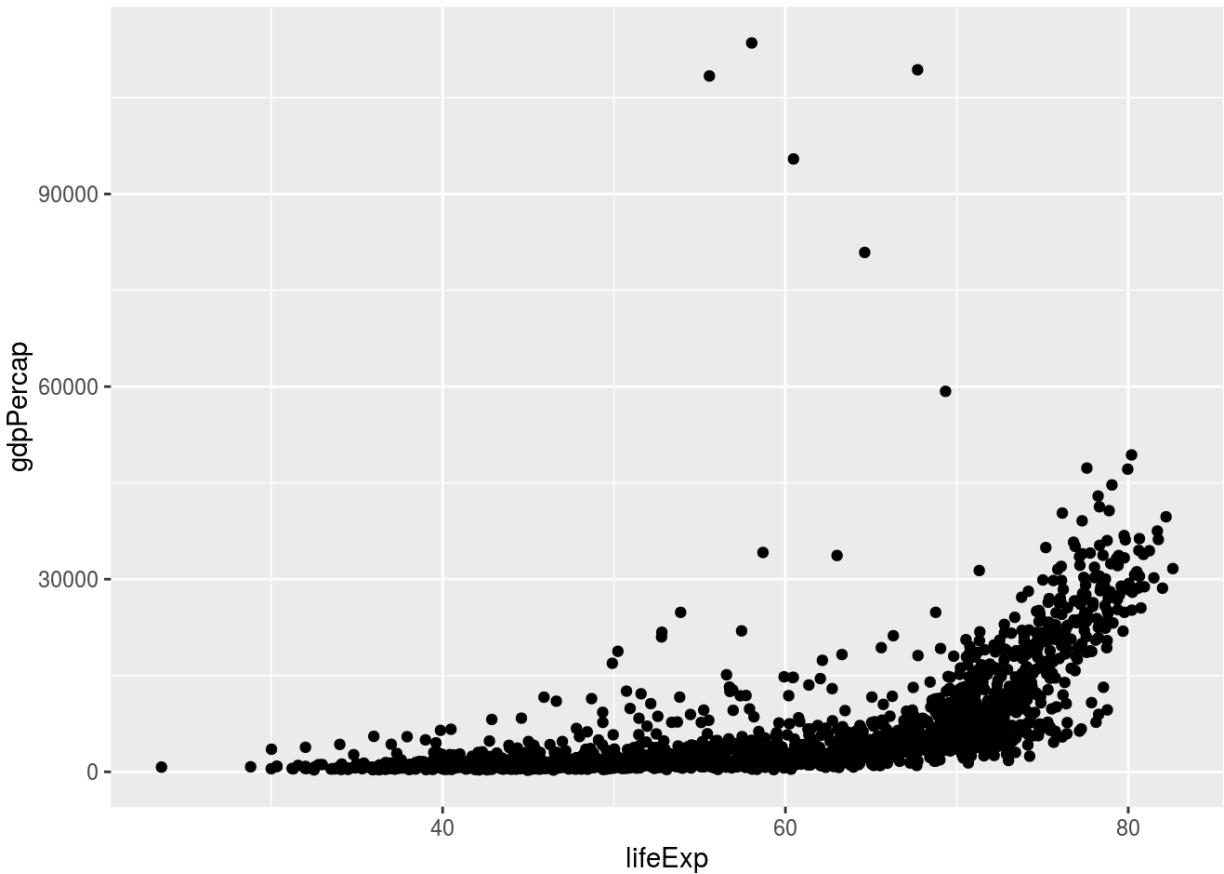
Plotar os nossos dados é uma das melhores formas para explorar os dados em si e as várias relações entre variáveis de forma rápida. Nessa parte vamos aprender sobre o pacote `ggplot2`, porque é o mais poderoso para criar gráficos com qualidade de publicação. `gplot2` é construído sobre a gramática de gráficos, a ideia de que qualquer parcela pode ser expressa a partir do mesmo conjunto de componentes: um grupo de ** dados ** set, um **sistema de coordenadas**, e um conjunto de **geoms** - a representação visual de pontos de dados. A chave para entender `ggplot2` é pensar em uma figura em camadas: assim como pode ser feito em um programa de edição de imagem como Photoshop, GIMP, Illustrator ou Inkscape. Vamos começar com um exemplo:

```
library(ggplot2)
ggplot(data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
  geom_point()
```



A primeira coisa a fazer é chamar a função `ggplot`. Esta função permite que o R entenda que estamos criando um novo plot, e qualquer dos argumentos que inserimos à função `ggplot` são as opções *globais* para o gráfico: elas se aplicam a toda parcela das camadas. Acima foram usados dois argumentos para `ggplot`. Primeiro, dizemos à função `ggplot` quais dados nós queremos mostrar na nossa figura (neste exemplo os dados gapminder que lemos anteriormente). Para o segundo argumento, informamos a função `aes`, que informa ao `ggplot` como as variáveis nos **dados** se disporão na figura. Neste caso os locais `x` e `y`. Aqui informamos ao `ggplot` que desejamos plotar a coluna “`lifeExp`” no eixo `x`, e a coluna “`gdpPercap`” no eixo `y`. Não é preciso informar explicitamente essas colunas ao `aes` (por exemplo, `x = gapminder[, "lifeExp"]`), isso ocorre porque o `ggplot` é esperto o suficiente para saber onde procurar os **dados** para essa coluna! Por si só, chamar pelo `ggplot` não é suficiente para desenhar uma figura: Precisamos dizer ao `ggplot` como queremos representar visualmente os dados, que fazemos adicionando uma nova camada `geom`. No exemplo, usamos `geom_point`, que diz ao `ggplot` que queremos representar visualmente a relação entre `x` e `y` como um scatterplot de pontos:

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



Desafio 1

Modifique o exemplo para que a figura mostre como a expectativa de vida se alterou ao longo do tempo:

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

Dica: o conjunto de dados gapminder tem uma coluna chamada “year”, que deve aparecer no eixo x.

Desafio 2

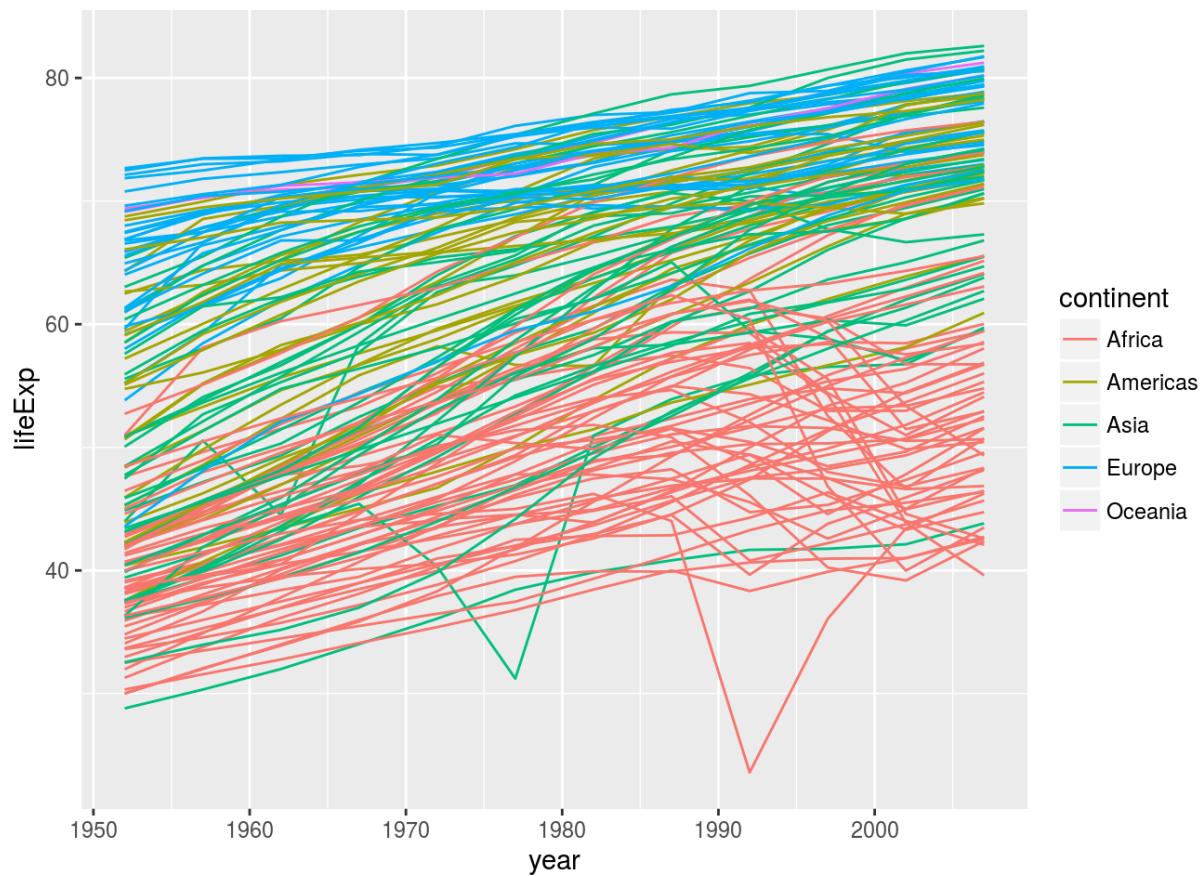
A função **aes** é usada para dizer ao gráfico **geom** qual as posições de cada ponto em **x** e **y**. Outra propriedade *estética* que podemos modificar é a *cor* dos pontos.

Modifique o código do desafio anterior para organizar por **cor** os dados da coluna “continent” e observe a tendências dos dados.

Camadas (Layers)

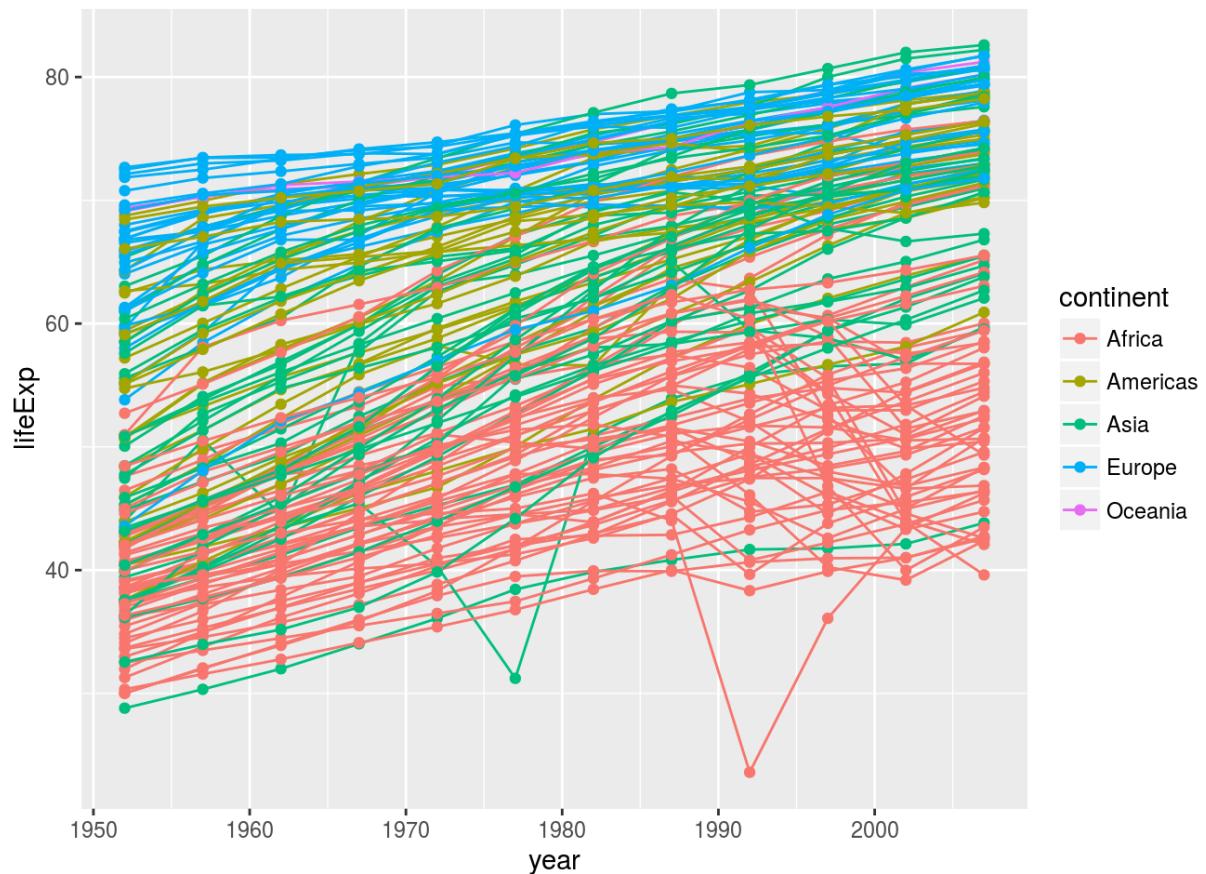
Usar um scatterplot provavelmente não é o melhor método para visualizar a mudança ao longo do tempo. Em vez disso, informaremos ao **ggplot** para visualizar os dados como um gráfico de linha:

```
ggplot(data = gapminder, aes(x=year, y=lifeExp, by=country, color=continent)) +
  geom_line()
```



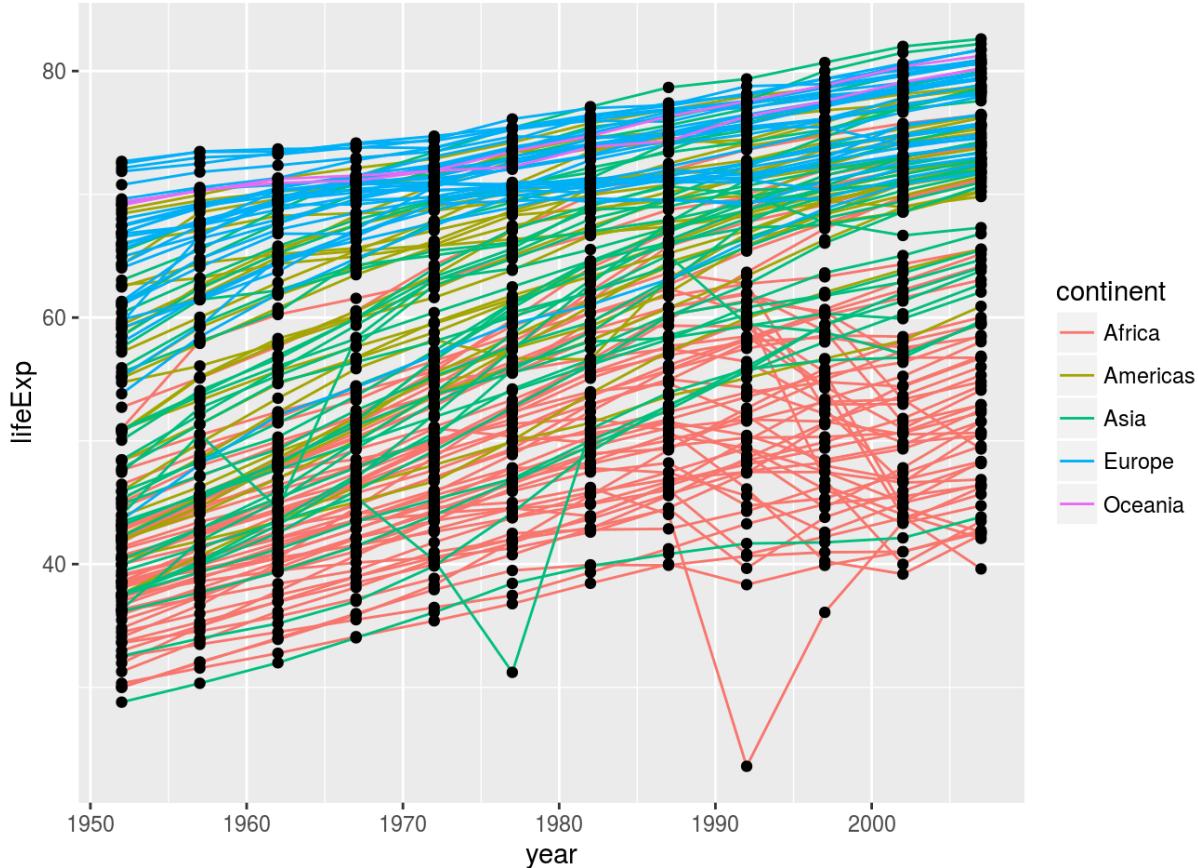
Em vez de adicionar uma camada `geom_point`, adicionamos uma camada `geom_line`. Adicionamos o `by` que diz ao `ggplot` para desenhar uma linha para cada país. Mas e se quisermos visualizar linhas e pontos no gráfico? É só adicionar outra camada ao gráfico:

```
ggplot(data = gapminder, aes(x=year, y=lifeExp, by=country, color=continent)) +
  geom_line() + geom_point()
```



É importante notar que cada camada é desenhada em cima da camada anterior. Neste exemplo, os pontos foram desenhados *em cima* das linhas. Abaixo, uma demonstração

```
ggplot(data = gapminder, aes(x=year, y=lifeExp, by=country)) +
  geom_line(aes(color=continent)) + geom_point()
```



Neste exemplo, o `estilo de cor` foi movido das opções globais de plotagem em `ggplot` para a camada `geom_line` para que ela não se aplique mais aos pontos. Agora podemos ver claramente que os pontos são desenhados em cima das linhas.

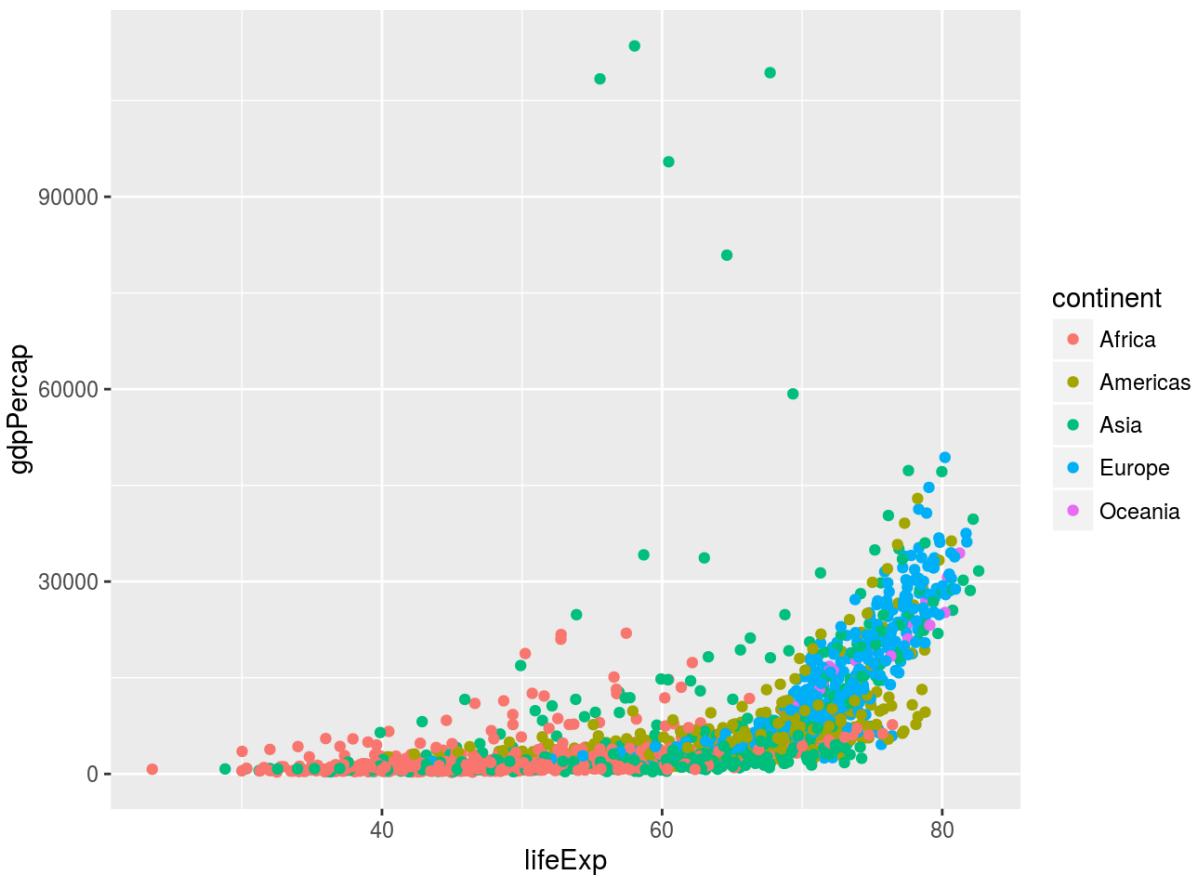
Desafio 3

Troque a ordem das camadas de ponto e linha do exemplo anterior e veja o que acontece.

Transformações e estatísticas

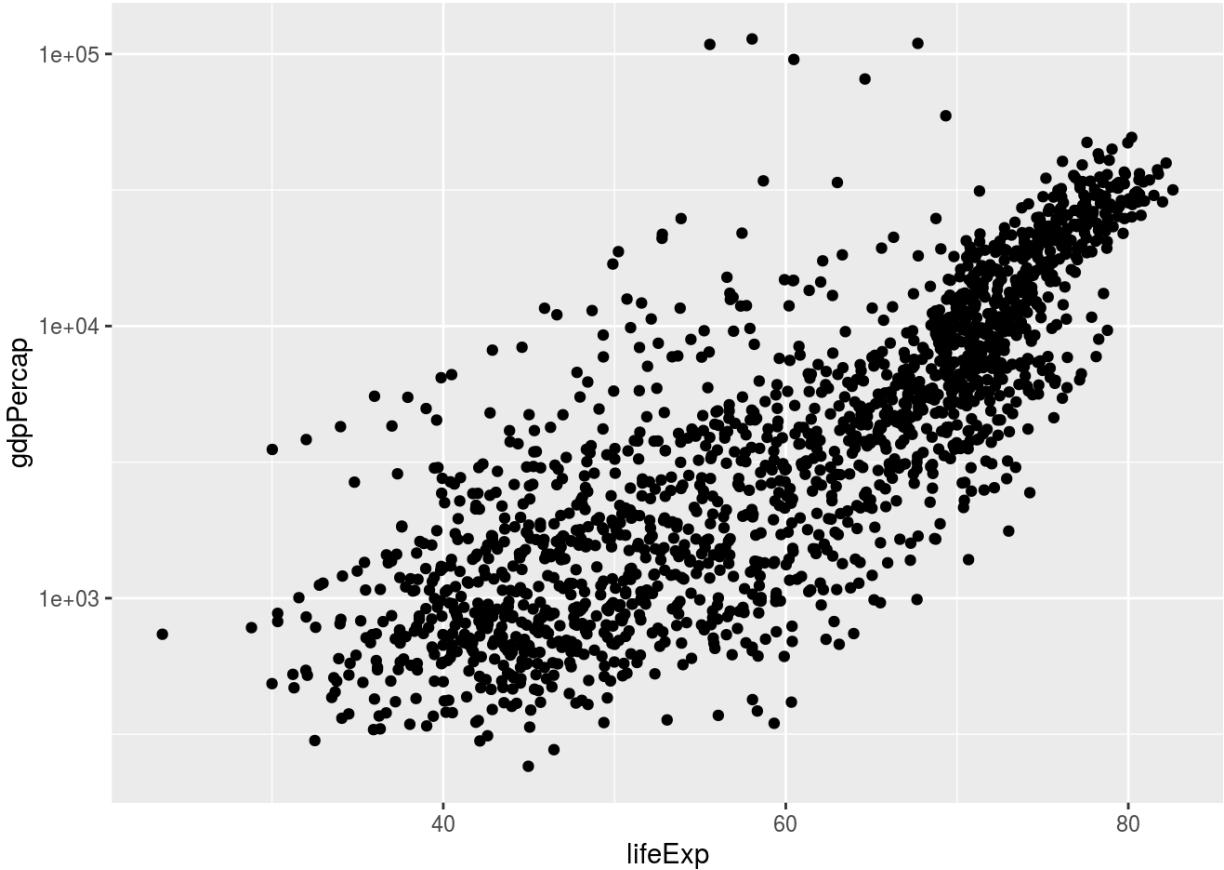
O `ggplot` também facilita a sobreposição de modelos estatísticos sobre os dados. Para demonstrar, voltaremos ao primeiro exemplo:

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



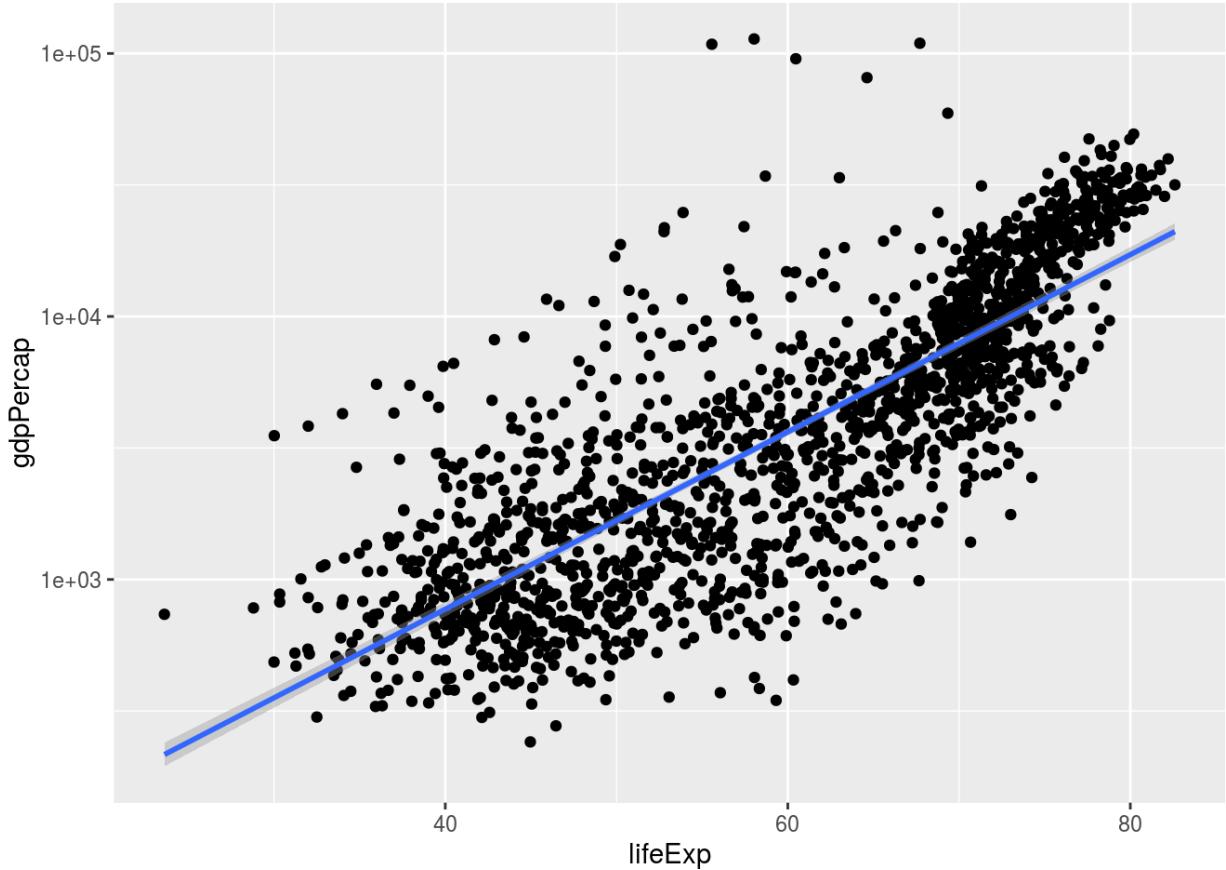
Atualmente, é difícil ver a relação entre os pontos devido a algum outlier acentuado no PIB per capito. Podemos alterar a escala de unidades no eixo y usando as funções `scale`. Essas funções controlam o mapeamento entre os valores de dados e os valores visuais de uma estética.

```
ggplot(data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
  geom_point() + scale_y_log10()
```



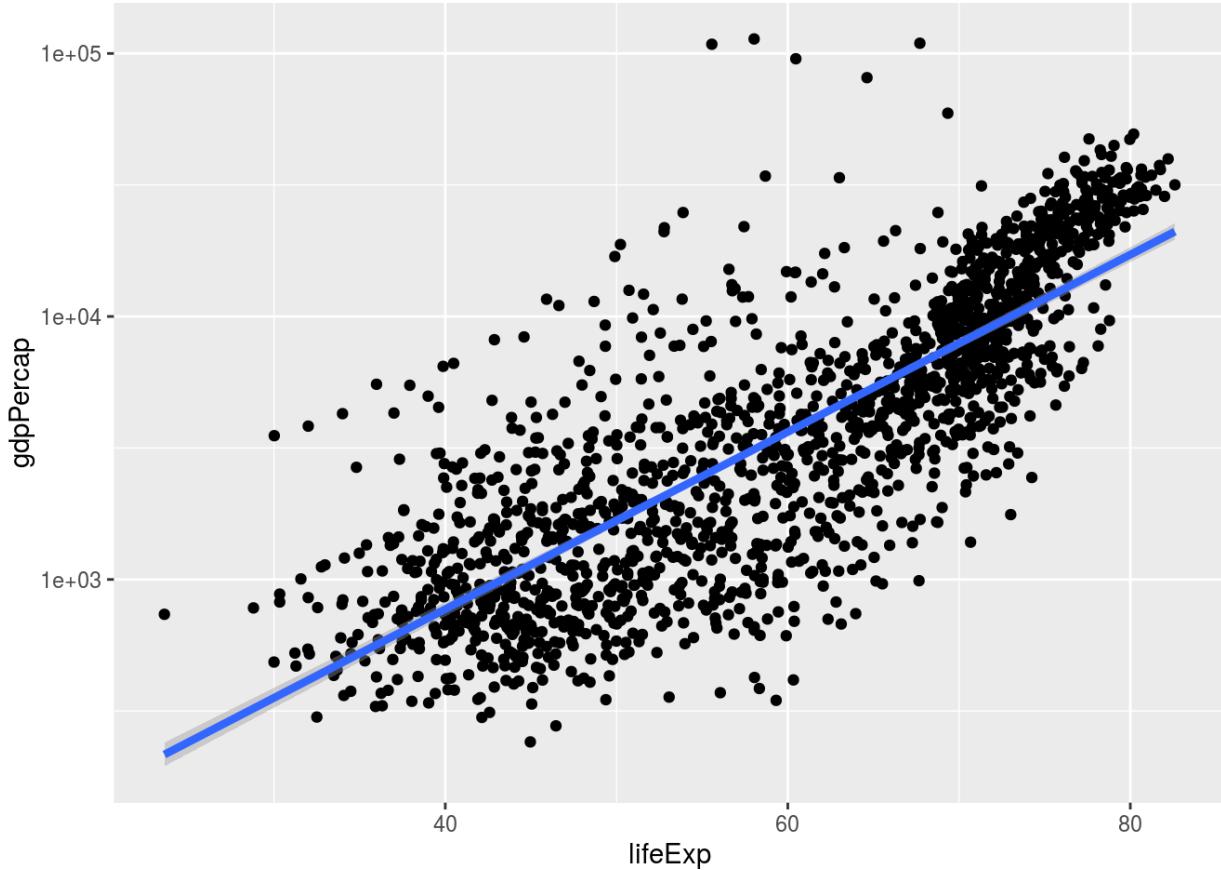
A função `log10` aplicou uma transformação aos valores da coluna de `gdpPercap` antes de renderizá-los no plot, de modo que cada múltiplo de 10 agora corresponde apenas a um aumento em 1 na escala transformada. Por exemplo: Um PIB per capita de 1.000 é agora 3 no eixo y, um valor de 10.000 corresponde a 4 no eixo y e assim por diante. Isto torna mais fácil visualizar a distribuição de dados no eixo y. Nós podemos ajustar uma relação simples aos dados adicionando uma outra camada, `geom_smooth`:

```
ggplot(data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
  geom_point() + scale_y_log10() + geom_smooth(method="lm")
```



Podemos deixar a linha mais fina *ajustando* a estética `size` na camada `geom_smooth`:

```
ggplot(data = gapminder, aes(x = lifeExp, y = gdpPercap)) +  
  geom_point() + scale_y_log10() + geom_smooth(method="lm", size=1.5)
```



Existem duas maneiras de especificar uma *estética*. Aqui nós definimos o seu **tamanho** (size=1.5), passando-a como argumento para `geom_smooth`, e antes usamos a função `aes` para definir um *mapeamento* entre variáveis de dados e a sua representação visual.

Desafio 4

Modifique a cor e o tamanho dos pontos na camada de ponto no exemplo anterior.

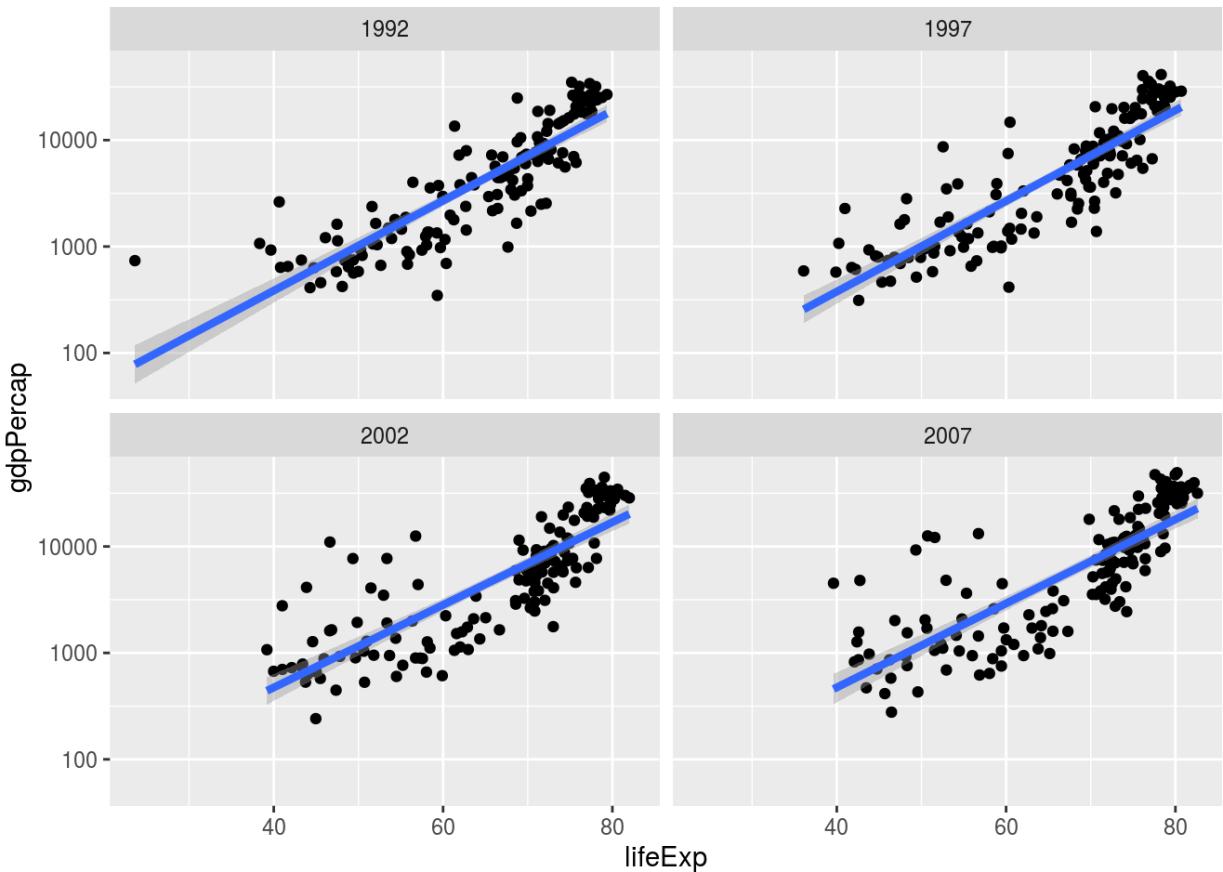
Dica: não use a função `aes`.

Figuras de múltiplos painéis

Anteriormente, visualizamos a mudança na expectativa de vida ao longo do tempo em todos os países em um único plot. Mas podemos dividir isso em vários painéis adicionando uma camada de painéis `facet`:

```
gapminder2 <- gapminder[gapminder$year > 1990,]

ggplot(data = gapminder2, aes(x = lifeExp, y = gdpPercap)) +
  geom_point() + scale_y_log10() + geom_smooth(method="lm", size=1.5) +
  facet_wrap(~ year)
```



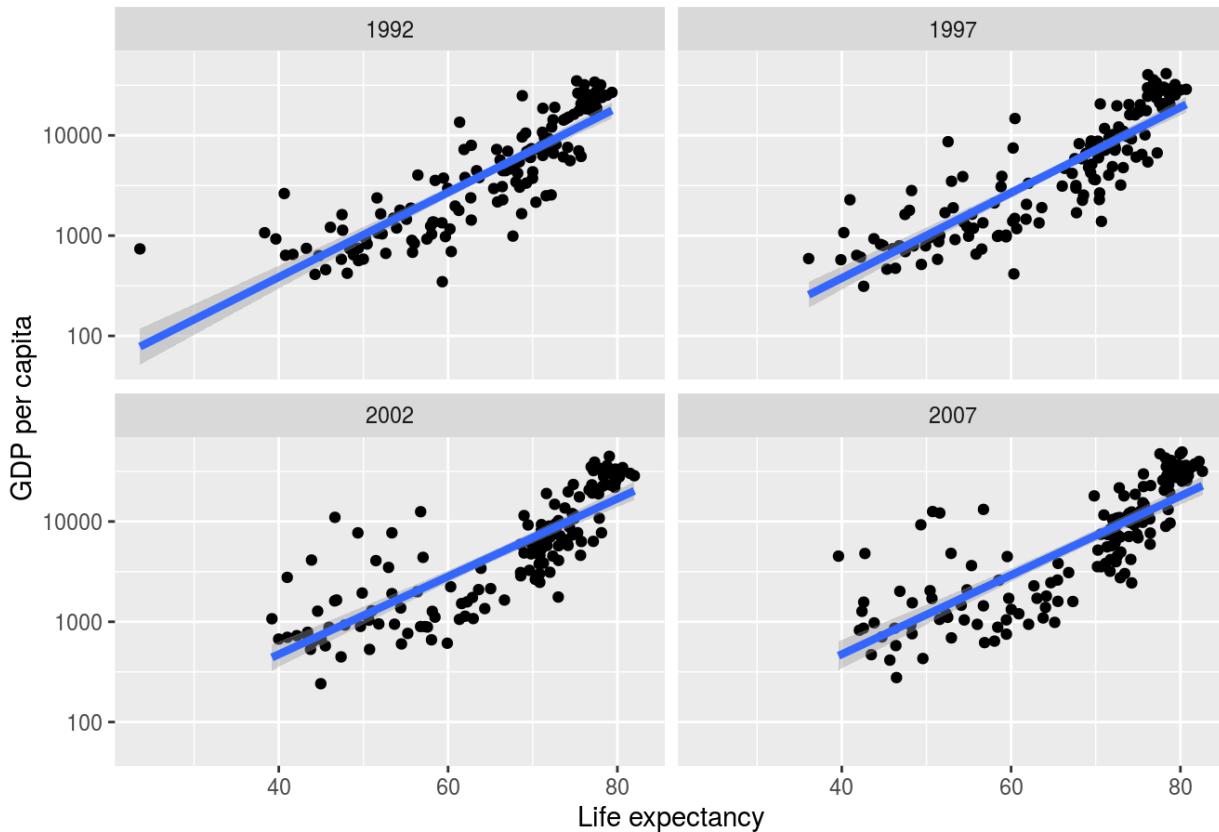
Uma “fórmula” foi inserida na camada `facet_wrap` como argumento, denotado pelo til (~). Isso informa ao R para desenhar um painel para cada valor único na coluna de país do conjunto de dados gapminder.

Modificando texto

Para limpar esta figura para uma publicação nós precisamos de mudar alguns elementos do texto. Podemos fazer isso adicionando um par de camadas diferentes. A camada `theme` controla o eixo do texto e o tamanho geral do texto. Também existem camadas especiais para alterar os rótulos dos eixos. Para alterar o título da legenda, precisamos usar camada `escales`.

```
ggplot(data = gapminder2, aes(x = lifeExp, y = gdpPercap)) +
  geom_point() + scale_y_log10() + geom_smooth(method="lm", size=1.5) +
  facet_wrap(~ year) +
  xlab("Life expectancy") + ylab("GDP per capita") + ggtitle("Figure 1")
```

Figure 1



Essa é só uma “palhinha” do que podemos fazer com o `ggplot2`. O RStudio fornece uma lista de “cola” muito útil, chamada cheat sheet com as diferentes camadas disponíveis, e uma extensa documentação está disponível no site do `ggplot2`. Se há uma dúvida de como mudar algo, uma pesquisa rápida no Google te levará a um relevante “pergunta e resposta” no Stack Overflow com códigos reutilizáveis para modificar!

Desafio 5

Crie um gráfico de densidade do PIB per capita, preenchido por continente.

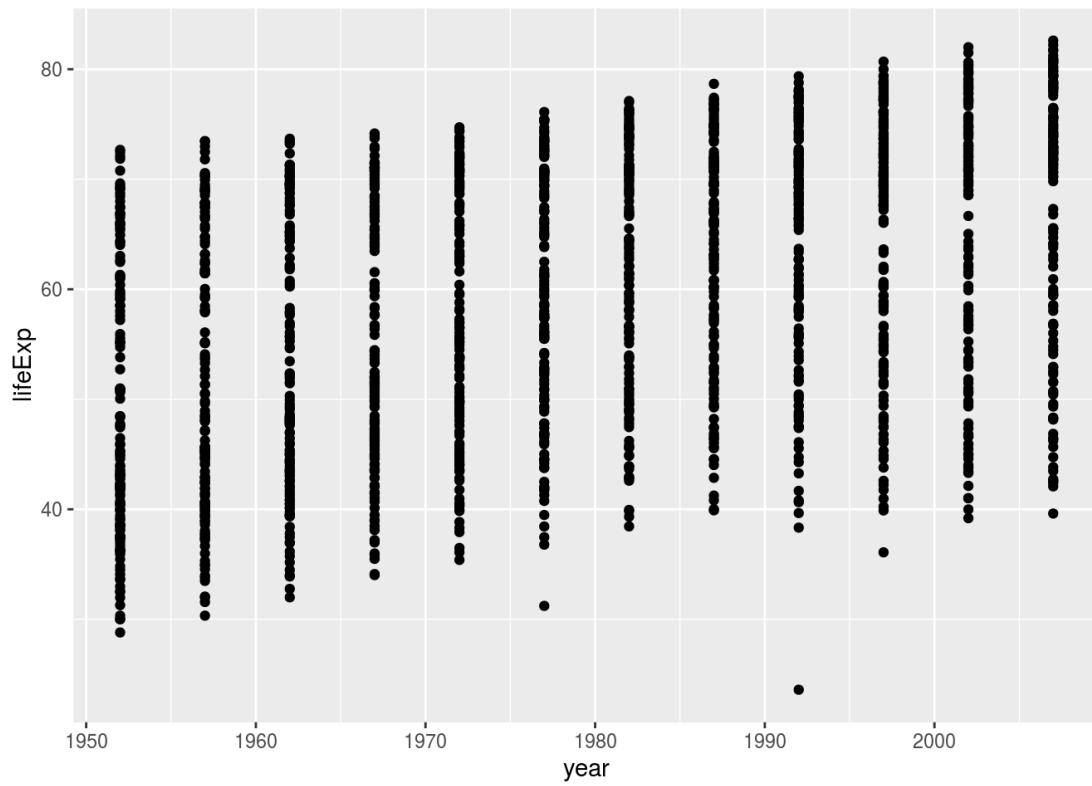
Avançado: - Transforme o eixo x para melhor visualizar a propagação de dados. - Adicione uma camada facet para exibir gráficos de densidade por ano.

Soluções de desafio

Solução do desafio 1

Modifique o exemplo para que a figura visualize como a expectativa de vida se altera ao longo do tempo:

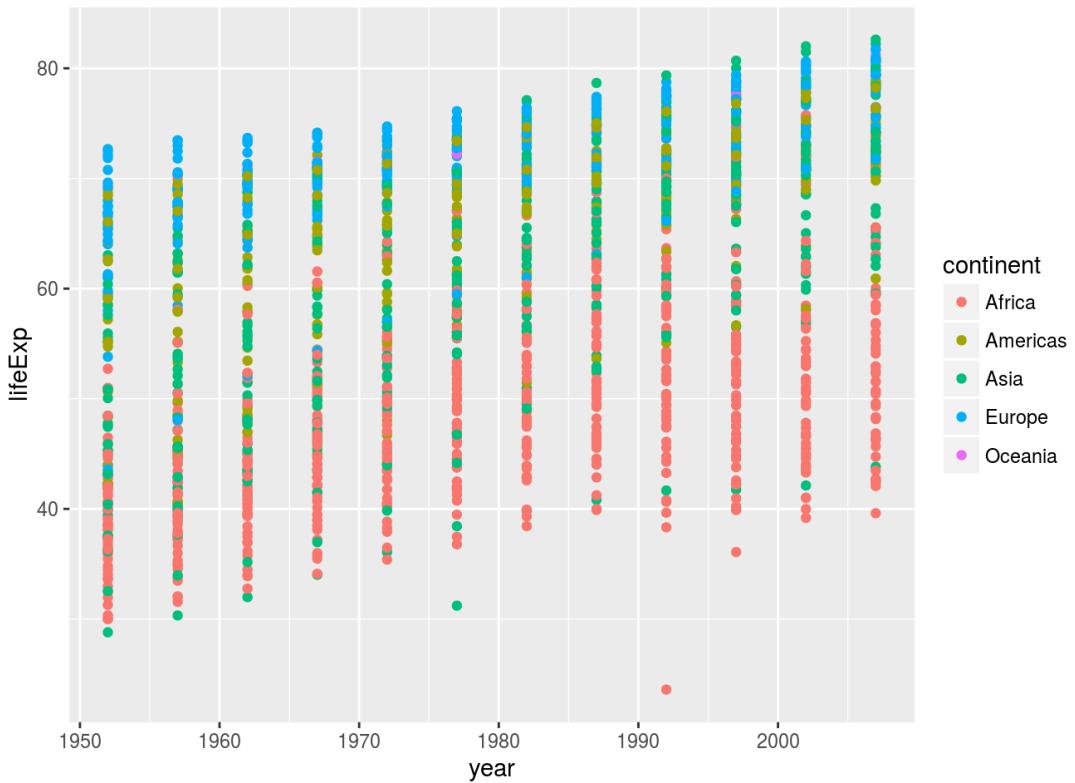
```
ggplot(data = gapminder, aes(x = year, y = lifeExp)) + geom_point()
```



Solução do desafio 2

Nos exemplos e desafio anteriores nós usamos a função `aes` para dizer ao scatterplot `geom` sobre as posições `x` e `y` de cada ponto. Outra propriedade *estética* que podemos modificar é a *cor* do ponto. Modifique o código do desafio anterior para **colorir** os pontos pela coluna “continente” e veja se há tendências nos dados?

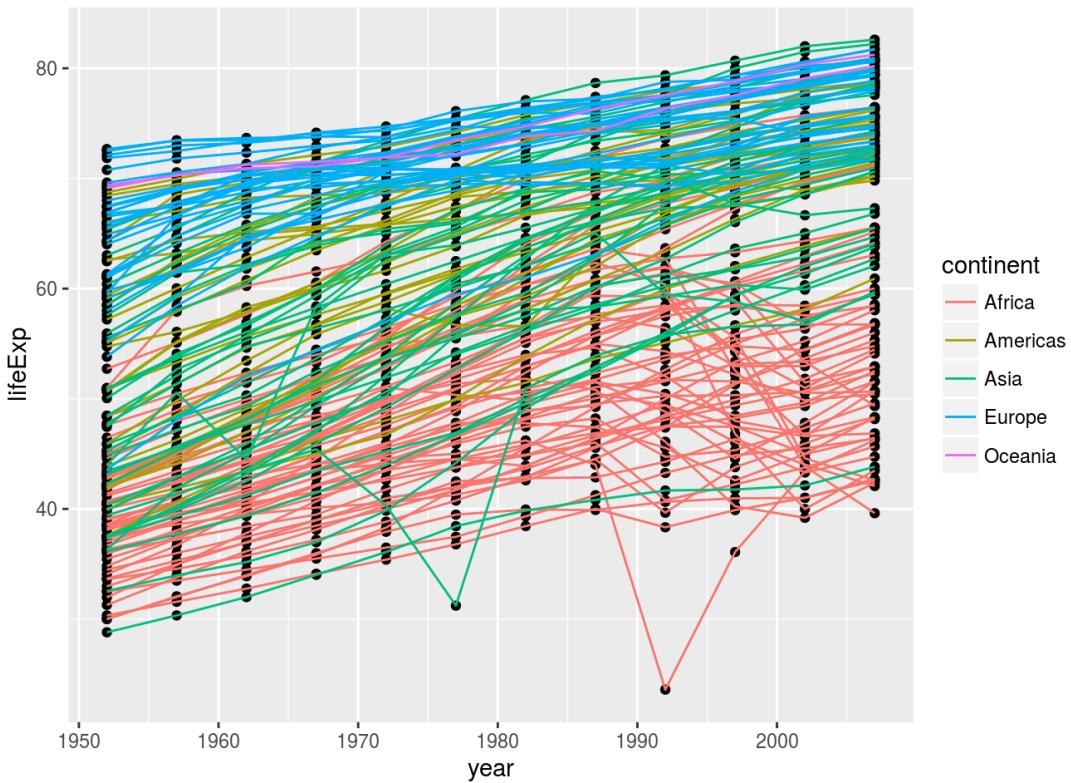
```
ggplot(data = gapminder, aes(x = year, y = lifeExp, color=continent)) +
  geom_point()
```



Solução do desafio 3

Troque a ordem das camadas de ponto e linha do exemplo anterior.

```
ggplot(data = gapminder, aes(x=year, y=lifeExp, by=country)) +
  geom_point() + geom_line(aes(color=continent))
```



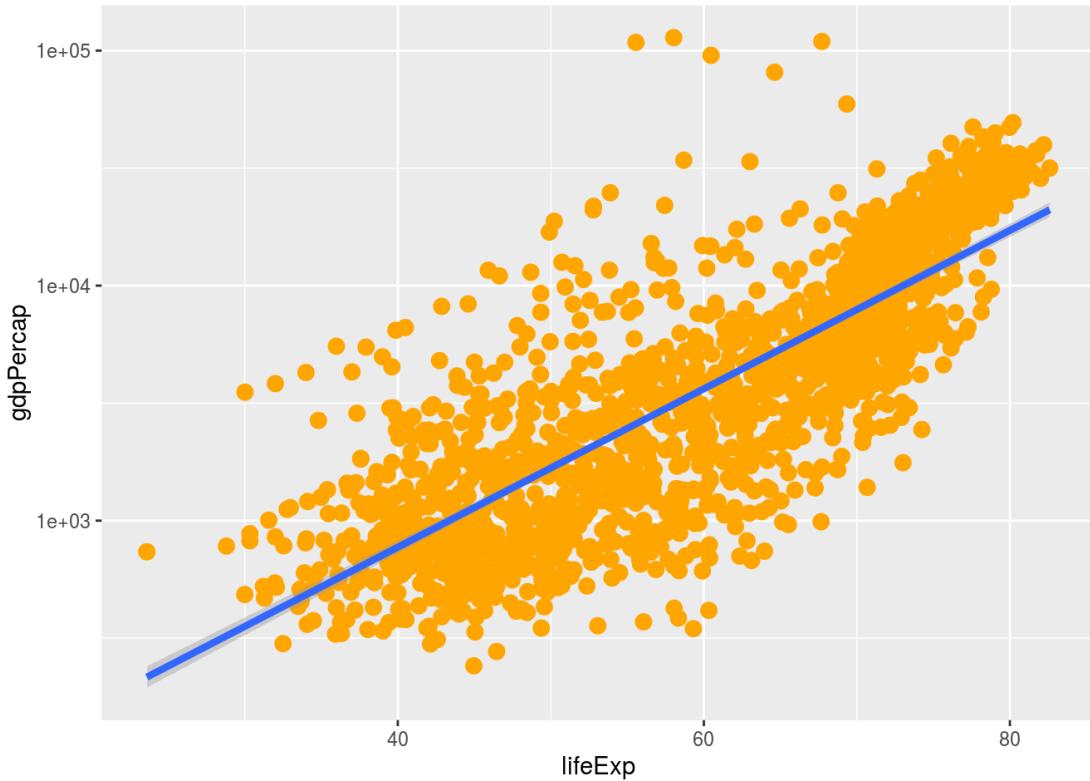
As linhas agora são desenhadas sobre os pontos!

Solução do desafio 4

Modifique a cor e o tamanho dos pontos na camada de ponto no exemplo anterior.

Dica: não use a função `aes`.

```
ggplot(data = gapminder, aes(x = lifeExp, y = gdpPercap)) +
  geom_point(size=3, color="orange") + scale_y_log10() +
  geom_smooth(method="lm", size=1.5)
```



Solução do desafio 5

Criar um gráfico de densidade do PIB per capita, preenchido por continente.

Avançado: - Transforme o eixo x para melhor visualizar a distribuição dos dados. - Adicione uma camada facet exibir gráficos de densidade por ano.

```
ggplot(data = gapminder, aes(x = gdpPercap, fill=continent)) +
  geom_density(alpha=0.6) + facet_wrap(~ year) + scale_x_log10()
```

