Apostila: introdução ao pacote dplyr

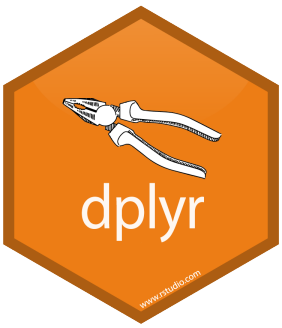
Me. Elisângela C. BiazattiDouglas Vinícius Jossivana Macedo

22 de outubro de 2019

Sumário

Capítulo 1

Prefácio



Este material foi elaborado com o proprósito de um minicurso, que tem como objetivo apresentar algumas ideias das funções básicas do pacote `dplyr`. Este material baseou-se em vários em livros, podemos usar um computador e um pouco de criatividade para explorar essas idéias em uma variedade de situações. Usamos R com o RStudio para fazer todo o nosso trabalho.

O livro R for data science é o mais indicado para aprender sobre o universo `tidyverse`. Foi usado o livro do Roger D. Peng R Programming for Data Science essencial para compreender as noções básicas do R. Nesse minicurso abordamos mais sobre a gramática das funções básicas do `dplyr` alguns exemplos e exercícios abordados.

1.1 Público-alvo

- Estudantes de estatística que desejam ganhar tempo nos trabalhos da faculdade;
- Acadêmicos com interesse em aprender análises e códigos mais legíveis em R.

1.2 Conteúdo:

```
- Primeiro dia (22/10): Breve introdução ao R, Tibbles x Data frames,
```

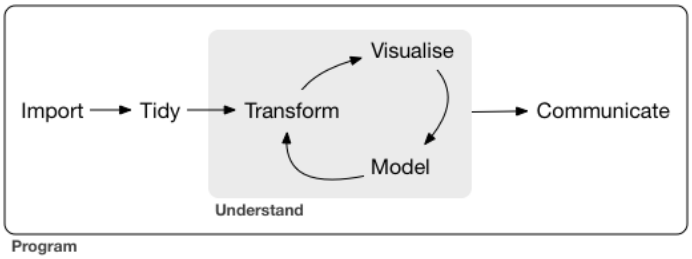
tidyr, select(), rename(), mutate(), exercícios;
- Segundo dia (23/10): filter(), arrange(), group_by(), summarise(), exercícios;
- Terceiro dia (24/10): .

Capítulo 2

Introdução

> "Existem apenas dois tipos de idiomas: os que as pessoas reclamam e os que ninguém usa". - Bjarne Stroustrup

O modelo típico de análise de dados é similar:



Primeiramente, você deve **importar** seus dados para o R. Significa que você pega os dados armazenados em um arquivo, banco de dados ou API da Web e carrega-os em um data frames no R.

Logo após, a ideia é **organizá**-los. Significa armazená-los de forma consistente.

Depois de arrumar os dados, o próximo passo é **transformá**-los. Significa restringir observações de interesse, criar novas variáveis

Depois de organizar os dados com as variáveis necessárias, existem dois mecanismos principais de geração de conhecimento: visualização e modelagem. Eles têm pontos fortes e fracos complementares, portanto qualquer análise real se repetirá entre eles várias vezes.

A **visualização** é uma atividade fundamentalmente humana. Uma boa visualização mostrará coisas que você não esperava, ou fará novas perguntas sobre os dados.

Modelos são ferramentas complementares para visualização. Depois de fazer suas perguntas suficientemente precisas, você pode usar um modelo para respondê-las.

O último passo da ciência de dados é a **comunicação**, uma parte absolutamente crítica de qualquer projeto de análise de dados. Não importa o quão bem seus

modelos e visualização levaram você a entender os dados, a menos que você também possa comunicar seus resultados a outras pessoas.

Ao redor de todas essas ferramentas está a programação. A programação é uma ferramenta transversal que você usa em todas as partes do projeto. Você não precisa ser um programador especialista para ser um cientista de dados, mas aprender mais sobre programação compensa, porque se tornar um programador melhor permite automatizar tarefas comuns e resolver novos problemas com maior facilidade.

2.1 Universo `tidyverse`

O tidyverse é uma coleção de pacotes R projetados para ciência de dados. Todos os pacotes compartilham uma filosofia de design, gramática e estruturas de dados subjacentes.

Os princípios fundamentais do `tidyverse` são:

- 1.Reutilizar estruturas de dados existentes;
- 2.Organizar funções simples usando o pipe;
- 3.Aderir à programação funcional;
- 4.Projetado para ser usado por seres humanos.

Assim como o processo típico do passo a passo apresentando anteriormente para análise de dados, o `tidyverse` é a ferramenta que o ajuda eficientemente a executar este processo.

```
library(tidyverse) #Carregar o pacote.
tidyverse_logo() #Logo
```

```
## * __  _    __   .    o          *  .
##  / /_(_)__/ /_ ___  _____ _______ ___
## / __/ / _  / // / |/ / -_) __(_-</ -_)
## \__/_/\_,_/\_, /|___/\__/_/ /___/\__/
##      *  . /___/      o      .       *
```

2.2 Porque devo aprender R e RStudio?

R é uma linguagem de programação estatística que vem passando por diversas evoluções e se tornando cada vez mais uma linguagem de amplos objetivos. Podemos entender o R também como um conjunto de pacotes e ferramentas

estatísticas, munido de funções que facilitam sua utilização, desde a criação de simples rotinas até análises de dados complexas, com visualizações bem acabadas.

Segue alguns motivos para aprender o R:

- É completamente gratuito e de livre distribuição;
- Curva de aprendizado bastante amigável, sendo muito fácil de se aprender;
- Enorme quantidade de tutoriais e ajuda disponíveis gratuitamente na internet;
- É excelente para criar rotinas e sistematizar tarefas repetitivas;
- Amplamente utilizado pela comunidade acadêmica e pelo mercado;
- Quantidade enorme de pacotes, para diversos tipos de necessidades;
- Ótima ferramenta para criar relatórios e gráficos.

Apenas para exemplificar-se sua versatilidade, esta apostila e os slides das aulas foram todos feitos em R.

A primeira coisa que você precisa fazer para iniciar o R é instalá-lo no seu computador. O R funciona em praticamente todas as plataformas disponíveis, incluindo os sistemas Windows, Mac OS X e Linux amplamente disponíveis.

- Para instalar o R, baixe a versão adequada para seu computador em: https://cloud.r-project.org/.

Uma nova versão principal do R sai uma vez por ano, e há 2 ou 3 versões menores a cada ano. É uma boa ideia atualizar regularmente. A atualização pode ser um pouco complicada, especialmente para as versões principais, que exigem a reinstalação de todos os seus pacotes.

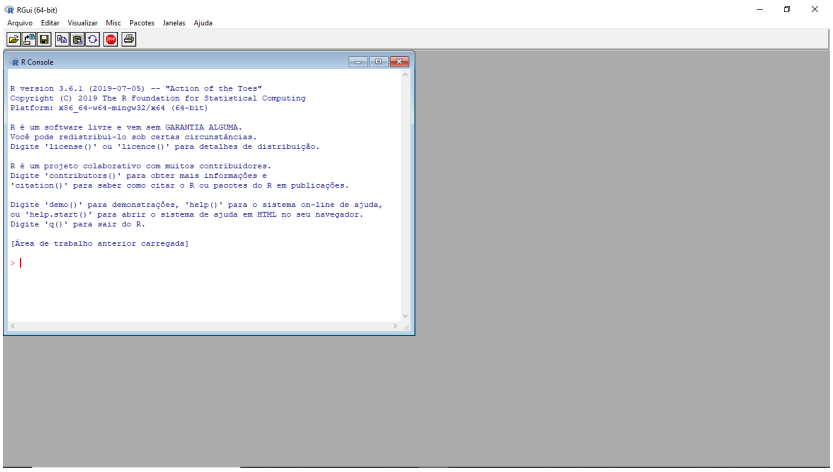
Há também um ambiente de desenvolvimento integrado (IDE) disponível para o R, construído pelo RStudio. IDE, do inglês **Integrated Development Environment** ou Ambiente de Desenvolvimento Integrado, é um programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de software com o objetivo de agilizar este processo. O RStudio é atualizado duas vezes por ano. Quando uma nova versão estiver disponível, o RStudio informará você.

- Para instalar o RStudio, baixe a versão adequada para seu computador em: https://www.rstudio.com/products/rstudio/download/.

Você pode ver como instalar o R e o RStudio aqui:

- Instalando o RStudio.

Após instalado, o R tem uma interface assim, com apenas o console para digitar comandos:

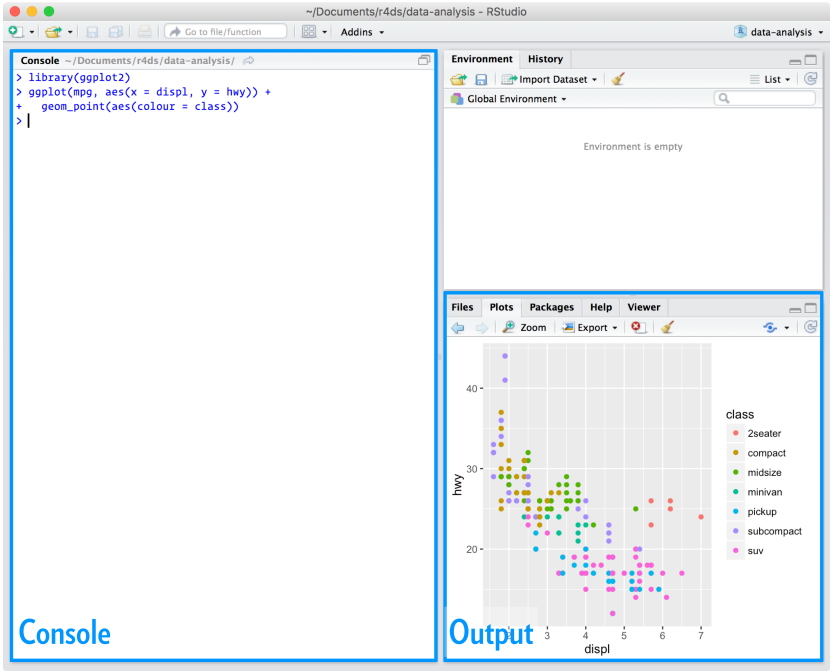


Experimente um comando: 2+2, cujo output é 4:

```
2 + 2
```

```
## [1] 4
```

E a interface do RStudio é dividida, inicialmente, em 3 partes:



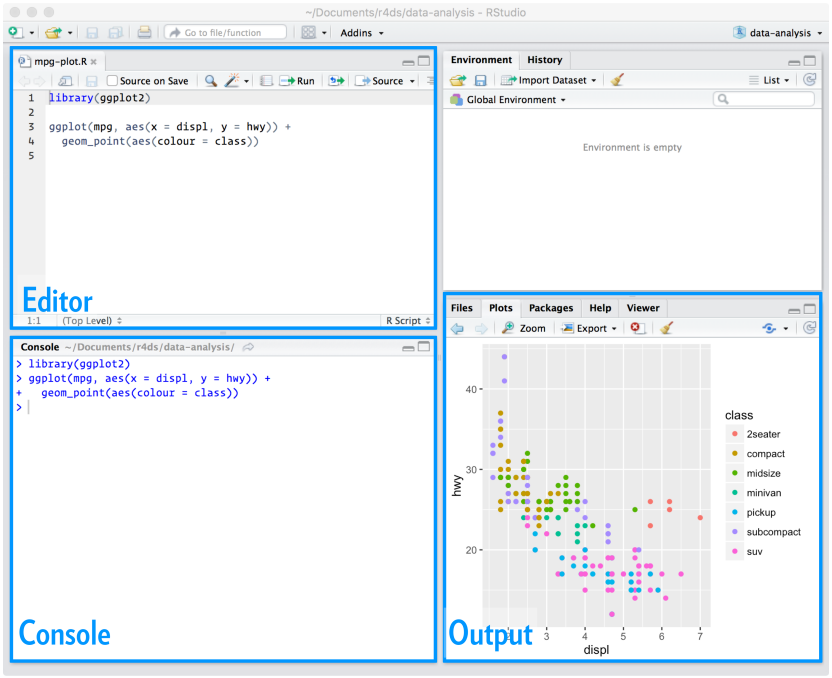
Do lado esquerdo fica o console, onde os comandos podem ser digitados e onde ficam os *outputs*.

No lado superior direito há duas abas:

-i) *Environment*, que é onde ficam armazendos os objetos criados, bases de dados importadas, etc; e

-ii) *History*, onde ficam o histórico dos comandos executados.

A forma mais eficiente e prática de usar o R ou o RStudio é através de um *script*. No RStudio, vá em *File* → *New File* → *R Script*. A interface agora fica dividida em 4 partes:



No *script* você pode digitar comandos a serem executados e também comentários.

2.3 Classes de Objetos R

R possui 5 classes básicas de objetos, também chamados de objetos "atômicas":

- character;
- numeric (real numbers);
- integer;
- complex;
- logical (True/False).

O tipo mais básico de objeto R é um vetor. Um vetor só pode conter elementos de uma mesma classe. Mas há uma exceção, que é uma lista. Uma lista é representada como um vetor, mas pode conter objetos de diferentes classes.

Características do vetor:

- Coleção ordenada de valores;
- Estrutura unidimensional.

Usando a função `c()` para criar vetores:

```
num <- c(10, 5, 2, 4, 8, 9)
num
```

```
## [1] 10  5  2  4  8  9
```

```
class(num) # para saber a classe do objeto, usamos a função class().
```

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

Por que `numeric` e não `integer`? Para forçar a representação de um número para inteiro é necessário usar o sufixo L.

```
x <- c(10L, 5L, 2L, 4L, 8L, 9L)
x
```

```
## [1] 10  5  2  4  8  9
```

```
class(x)
```

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

Note que a diferença entre `numeric` e `integer` também possui impacto computacional, pois o armazenamento de números inteiros ocupa menos espaço na memória. Dessa forma, esperamos que o vetor x acima ocupe menos espaço na memória do que o vetor num, embora sejam aparentemente idênticos. Usamos a função `object.size()` fornece uma estimativa da memória que está sendo usada para armazenar um objeto R.Veja:

```
object.size(num)
```

```
## 96 bytes
```

```
object.size(x)
```

```
## 80 bytes
```

A diferença pode parecer pequena, mas pode ter um grande impacto computacional quando os vetores são formados por milhares ou milhões de números

2.4 Atributos

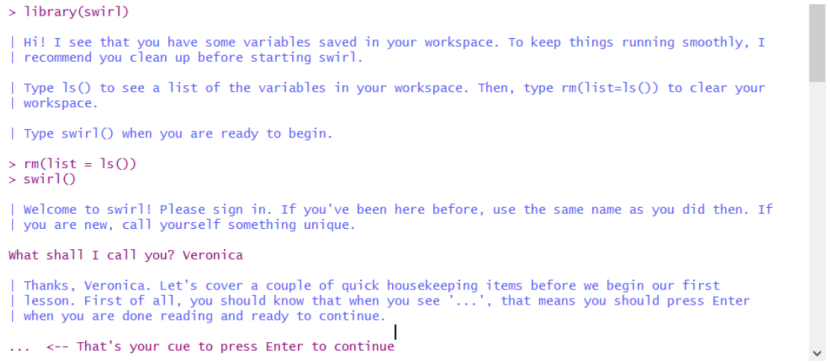
Os objetos R podem ter atributos, como metadados para o objeto. Esses metadados podem ser muito úteis, pois ajudam a descrever o objeto. Por exemplo, nomes de colunas em um quadro de dados ajudam a nos dizer quais dados estão contidos em cada uma das colunas. Alguns exemplos de atributos de objeto R são:

- nomes, dimnames;
- dimensões (por exemplo, matrizes, matrizes);
- classe (por exemplo, inteiro, numérico);
- comprimento;
- outros atributos/metadados definidos pelo usuário.

Os atributos de um objeto (se houver) podem ser acessados usando a função `attributes()`. Nem todos os objetos R contêm atributos; nesse caso, a `attributes()` retorna `NULL`.

2.5 *swirl*

O *swirl* é um pacote do R construído para transformar o console em uma ferramenta interativa para aprender R. *swirl* ensina programação de R e ciência de dados interativamente, no seu próprio ritmo e diretamente no console do R. Para entender melhor o projeto, veja http://swirlstats.com/ e http://swirlstats.com/students. Nestes endereços são dados os detalhes sobre como usar o *swirl*. Uma vez intalado e carregado o pacote, você é levado a efetuar tarefas:



O *swirl* dá acesso às tarefas de cursos de R que estão disponíveis também no Coursera, como o *R Programming*: *The basics of programming in R*, em https://pt.coursera.org/learn/r-programming. Além deste, estão disponíveis no *swirl*: *Regression Models: The basics of regression modeling in R, Statistical Inference: The basics of statistical inference in R, e Exploratory Data Analysis: The basics of exploring data in R.*

Capítulo 3

Tibbles x Data frames

> "Famílias felizes são todas iguais; toda família infeliz é infeliz à sua maneira." – Leo Tolstoi

O que é um *tibble*? *Tibbles* são similares aos *data frames*, porém diferentes em dois aspectos: **impressão** e **indexação**

Na impressão no console, os *tibbles* apresentam apenas as dez primeiras linhas e todas as colunas que cabem na tela, tornando mais fácil o trabalho com grandes volumes de dados. Além disso, cada coluna apresenta o seu tipo, algo semelhante ao apresentado quando utilizamos a função `str()`. A segunda diferença, não menos importante, é a forma de indexação. Para indexar um `tibble` devemos utilizar o nome completo da variável que desejamos. Caso contrário, ocorrerá um erro.

Ainda sobre a indexação, sempre que indexarmos um `tibble` usando `[`, o resultado será outro `tibble`. Usando `[[` o resultados será um vetor.

Em síntese, *data frames* são tabelas de dados. Em seu formato, são bem parecidos com as matrizes, no entanto, possuem algumas diferenças significativas. Podemos idealizar os *data frames* como sendo matrizes em que cada coluna pode armazenar um tipo de dado diferente. Logo, estamos lidando com um objeto bem mais versátil do que as matrizes e os vetores.

Uma das funções básicas mais importantes para começarmos a trabalhar com *data frames* é a `str()`. Essa função dá uma visão clara da estrutura do nosso objeto, bem como informa os tipos de dados existentes.

A função `View()` chama um visualizador de dados no estilo de planilhas em um objeto R. Semelhante a planilha do excel.

```
View(x, title)
```

Os argumentos da função são: `x` um objeto do R que pode ser coagido a um quadro de dados. E `title`, título para a janela do visualizador. O padrão é o nome de x prefixado.

3.1 O que é Dados organizados?

> "Os conjuntos de dados organizados são todos iguais, mas todos os conjuntos de dados confusos são confusos à sua maneira." – Hadley Wickham

Costuma-se dizer que 80% da análise de dados é gasta no processo de limpeza e preparação os dados (Dasu e Johnson 2003). A preparação de dados não é apenas um primeiro passo, mas deve ser repetidos muitos ao longo da análise, à medida que novos problemas surgem ou novos dados são coletados.Você vai precisar instalar os pacotes `tidyr`, `devtools` e `DSR`. Para instalar `tidyr` e `devtools`, abra o RStudio e execute o comando:

```
install.packages(c("tidyr", "devtools"))
```

`DSR` é uma coleção de conjuntos de dados. Para instalar DSR, execute o comando:

```
devtools::install_github("garrettgman/DSR")
```

Os dados tabulares podem ser organizados de várias maneiras. Os conjuntos de dados abaixo mostram os mesmos dados organizados de quatro maneiras diferentes, sendo que possuem as mesmas variáveis: país, ano, população e casos. Mas cada conjunto organiza os valores em forma de layout diferente. Vejamos essas tabelas de dados seguintes:

```
library(DSR)
# Primeiro conjunto de dados.
table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <fct>       <int>  <int>      <int>
## 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
```

```
# Segundo conjunto de dados.
table2
```

```
## # A tibble: 12 x 4
##    country      year key            value
##    <fct>       <int> <fct>          <int>
##  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
```

```
# Terceiro conjunto de dados.
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
##   <fct>       <int> <chr>
## 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
```

O último conjunto de dados é uma coleção de duas tabelas.

```
# Quarto conjunto de dados.
table4 # cases
```

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

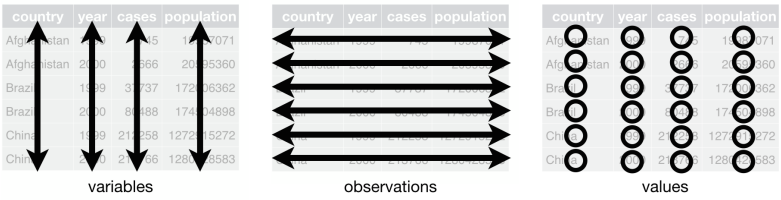
```
table5 # population
```

```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
##   <fct>            <int>      <int>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

R segue um conjunto de convenções que tornam um layout de dados tabulares muito mais fácil de trabalhar do que outros. Seus dados serão mais fáceis de trabalhar no R se seguirem três regras:

- 1.Cada variável no conjunto de dados é colocada em sua própria coluna;
- 2.Cada observação é colocada em sua própria linha;
- 3.Cada valor é colocado em sua própria célula.

Os dados que satisfazem essas regras são conhecidos como dados organizados. Observe que `table1` são dados organizados.



Essas três regras estão inter-relacionadas, porque é impossível satisfazer apenas duas das três. Essa inter-relação leva a um conjunto ainda mais simples de instruções práticas:

- Coloque cada conjunto de dados em um `tibble`;
- Coloque cada variável em uma coluna.

Em `table1`, cada variável é colocada em sua própria coluna, cada observação em sua própria linha e cada valor em sua própria célula.

Por que garantir que seus dados estejam organizados? Existem duas vantagens principais:

- Há uma vantagem geral em escolher uma maneira consistente de armazenar dados. Se você possui uma estrutura de dados consistente, é mais fácil aprender as ferramentas que funcionam com ela porque elas têm uma uniformidade subjacente;
- Há uma vantagem específica em colocar variáveis em colunas porque permite que a natureza vetorizada de R seja eficiente. Você aprenderá nas funções `mutate` e `summary`, a maioria das funções R internas trabalha com vetores de valores. Isso faz com que a transformação de dados organizados pareça particularmente natural.

3.2 Operador *pipe* %>%

O pacote `magrittr` tem dois objetivos: diminuir o tempo de desenvolvimento e melhorar a legibilidade e a manutenção do código. Para começar a utilizar o *pipe*, instale e carregue o pacote `magrittr`.

```
install.packages("magrittr")
```

```
library(magrittr)
```

Tubulação básica:

- x %>% *f* é equivalente a *f(x)*;
- x %>% *f(y)* é equivalente a *f(x, y)*;
- x %>% *f* %>% *g* %>% *h* é equivalente a *h(g(f(x)))*.

O operador do **pipeline** %>% é muito útil para reunir várias funções em uma sequência de operações. Os tubos são uma ferramenta poderosa para expressar claramente as operações. O **pipe**, %>% vem do pacote `magrittr` de Stefan Milton Bache. Pacotes no `tidyverse` carregam %>% automaticamente, para que normalmente não carregue o magrittr explicitamente. Observe abaixo que toda vez que desejamos aplicar mais de uma função, a sequência é ocultada em uma sequência de chamadas de funções aninhadas difíceis de ler, ou seja:

third(second(first(x)))

Esse aninhamento não é uma maneira natural de pensar em uma sequência de operações. O %>% permite que você encadeie operações da esquerda para a direita, ou seja:

first(x) %>% second() %>% third()

Por exemplo:

```
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)

round(exp(diff(log(x))), 1)
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

```
#Com a ajuda de `%>%`:

x %>%
  log() %>%
  diff() %>%
  exp() %>%
  round(1)
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

Em resumo, aqui estão quatro razões pelas quais você deve usar tubos ou *pipe* no R:

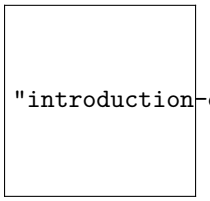
- Estruturara a sequência de suas operações de dados da esquerda para a direita, ao contrário de dentro para fora;
- evita chamadas de função aninhadas;
- minimiza a necessidade de variáveis locais e definições de funções;
- facilita a adição de etapas em qualquer lugar da sequência de operações.

Mesmo sendo %>% o operador de tubulação (principal) do pacote `magrittr`, existem alguns outros operadores que fazem parte do mesmo pacote:

Ao trabalhar com tubos mais complexos, às vezes é útil chamar uma função por seus efeitos colaterais. Talvez você queira imprimir o objeto atual, plotá-lo ou salvá-lo em disco. Muitas vezes, essas funções não retornam nada, efetivamente encerrando o pipe.

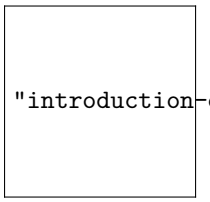
Para contornar esse problema, usar-se o tubo "tee". `%T>%` funciona como `%>%` exceto que retorna o lado esquerdo em vez do lado direito. É chamado de "tee" porque é como um tubo em forma de T literal.

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()
```




```
##  NULL
```

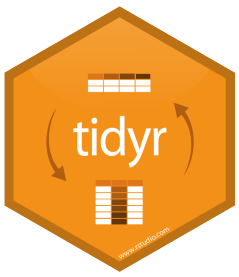
```
rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
```




```
##  num [1:50, 1:2] 0.339 -0.918 -1.039 0.182 -0.571 ...
```

Para mais informações sobre o *pipe*, outros operadores relacionados e exemplos de utilização, visite a página Ceci n'est pas un pipe. Ou consulte a vinheta do pacote `vignette("magrittr")`.

3.3 Breve Introdução ao `tidyr`



O pacote tidyr tem como principal objetivo transformar um data frame para o formato tidy, ou limpo.

De acordo com as regras ditas anteriormente, um dado limpo/organizado é aquele com formato *long*, ou seja, com mais linhas. O outro formato é chamado

de *wide*, com mais colunas. No caso deste exemplo, ano é uma variável, logo é necessário existir uma coluna com os valores de ano. O valor relacionado a UF naqueles anos também é outra variável, então precisa de uma coluna pra representá-lo. Além disso, a própria UF precisa de uma coluna.

O `tidyr` possui duas funções principais:

`gather` = *amontoar*: Transforma um `tibble` *wide* em *long*, ou seja, transforma os dados no formato *tidy*.

`spread` = *esparramar*: Transforma um `tibble` *long* em *wide*, ou seja, transforma dados que estão no formato *tidy* em formato não *tidy*.

Além disso, existem duas funções que podem ser importantes na nossa análise: `separate` e `unite`, que separa uma coluna em duas e vice versa.

Utilizando o conjunto de dados `storms` do pacote `EDAWR`, que descreve a velocidades máximas do vento para seis furacões no Atlântico.

```
library(tidyr)
library(EDAWR)
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

Como você pode ver, toda variável (nome da tempestade, vento, pressão, data) tem sua própria coluna e toda observação é salva em sua própria linha.

Como você pode ver, toda variável (nome da tempestade, vento, pressão, data) tem sua própria coluna e toda observação é salva em sua própria linha.

```
cases
```

```
##   country  2011  2012  2013
## 1      FR  7000  6900  7000
## 2      DE  5800  6000  6200
## 3      US 15000 14000 13000
```

O conjunto de dados `cases` têm três variáveis: o código do país em cada linha, o ano em cada coluna e uma contagem para cada combinação de linha e colunas. Como você pode ver, essa estrutura não está em conformidade com as características de um conjunto de dados organizado mencionado acima.

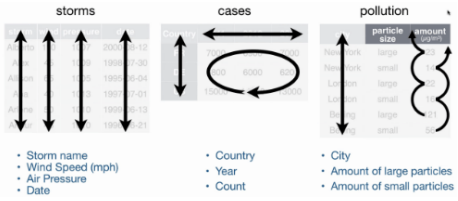
Vamos dar uma olhada em outro exemplo de um conjunto de dados não arrumado chamado poluição:

```
pollution
```

```
##       city  size amount
## 1 New York large     23
## 2 New York small     14
```

```
## 3   London large     22
## 4   London small     16
## 5  Beijing large    121
## 6  Beijing small     56
```

Aqui temos três variáveis: nomes de cidades, quantidade de partículas pequenas e quantidade de partículas grandes que cada cidade possui (indice de qualidade do ar). Novamente, isso não está em conformidade com as características de um conjunto de dados organizado. A próxima figura resume graficamente a estrutura dos três conjuntos de dados:



3.4 Função `gather()`

Se procurarmos a documentação para esta função, podemos encontrar: "usa o `gather()` quando percebe que possui colunas que não são variáveis." Esta declaração se aplica ao conjunto de dados `cases` descrito acima. Então, vamos tentar arrumar usando a `gather()`. A função `gather()` retorna um `tibble` com duas colunas, por padrão, isso se não inserirmos nenhum parâmetro além do `tibble`.

```
tidy.cases <- gather(cases, "year", "n", 2:4)
```

O resultado é um quadro de dados organizado com três colunas, onde cada coluna representa uma variável e cada obeservação é salva em sua própria linha.

3.5 Função `spread()`

A segunda função principal do `tidyr` é `spread()`. Esta função pega dados que estão em um formato de valor-chave e retorna um formato retangular de célula organizada. Isso pode parecer confuso, então vamos aplicar esta função no quadro de dados de poluição para ilustrar sua funcionalidade:

```
tidy.pollution <- spread(pollution, size, amount)
```

Como você pode ver, `spread()` reestrutura o quadro de dados removendo linhas redundantes sem perder nenhuma informação.

Duas outras funções úteis são `unite()` e `separate()`. Para mostrar seu uso, vamos olhar o quandro de dados `storms` novamente:

```
storms
```

```
##     storm wind pressure       date
## 1 Alberto  110     1007 2000-08-03
## 2    Alex   45     1009 1998-07-27
```

```
## 3 Allison   65     1005 1995-06-03
## 4     Ana   40     1013 1997-06-30
## 5  Arlene   50     1010 1999-06-11
## 6  Arthur   45     1010 1996-06-17
```

`separate()` pode ser usado para separar uma coluna em várias outras colunas usando um separador. Digamos, por exemplo, que, em vez do formato de data AAAA-MM-DD que esta no `storms` e em uma única coluna, desejemos três colunas separadas: uma com o ano, uma com o mês e outra com o dia. Isso pode ser alcançado com o seguinte comando:

```
storms.sep <- separate(storms, date, c("year", "month", "day"), sep="-")
```

`unite()` faz exatamente o oposto. Ele une várias colunas em uma única coluna. Isso pode ser demonstrado usando nosso novo quadro de dados:

```
unite(storms.sep, "date" , 4:6 , sep ="-")
```

```
## # A tibble: 6 x 4
##   storm    wind pressure date
##   <chr>   <int>    <int> <chr>
## 1 Alberto   110     1007 2000-08-03
## 2 Alex       45     1009 1998-07-27
## 3 Allison    65     1005 1995-06-03
## 4 Ana        40     1013 1997-06-30
## 5 Arlene     50     1010 1999-06-11
## 6 Arthur     45     1010 1996-06-17
```

3.5.1 Exercícios

- 1.Qual a diferença entre uma matriz e um data frame no R?
- 2.Os data frames podem ser indexados com a mesma sintaxe utilizada para matrizes?
- 3.Qual função básica que utilizamos para verificar a estrutura dos dados de um data frame?

3.5.2 Exercícios

- 1. Reescreva a expressão abaixo utilizando o `%>%`.

  round(mean(sum(1:10)/3), digits = 1)

Dica: utilize a função `magrittr::divide_by()`. Veja o help da função para mais informações.

- 2. Reescreva o código abaixo utilizando o `%>%`.

  x <- rnorm(100) x.pos <- x[x>0] media <- mean(x.pos) saida <- round(media, 1)

- 3. Sem rodar, diga qual a saída do código abaixo. Consulte o `help` das funções caso precise.

  2 %>% add(2) %>% c(6, NA) %>% mean(na.rm = T) %>% equals(5)

Capítulo 4

Manipulando Data Frames com `dplyr`

Entendamos a manipulação de dados como o ato de transformar, reestruturar, limpar, agregar e juntar os dados. Para se ter uma noção da importância dessa fase, alguns estudiosos da área de Ciência de Dados costumam afirmar que 80% do trabalho é encontrar uma boa fonte de dados, limpar e preparar os dados, sendo que os 20% restantes seriam o trabalho de aplicar modelos e realizar alguma análise propriamente dita.

4.1 O Pacote `dplyr`

O pacote `dplyr` foi desenvolvido por Hadley Wickham, cientista chefe do RStudio. É uma versão otimizada do pacote `plyr`. O pacote `dplyr` não fornece nenhuma funcionalidade "nova" ao R, pois já é feito com base no R, mas simplifica **bastante** a funcionalidade no R.

Uma contribuição importante do `dplyr` é que ele fornece uma "gramática" (em particular, verbos) para manipulação

4.2 Gramática do `dplyr`

Alguns dos principais "verbos" básicos de tablea única fornecidos pelo `dplyr` são:

-`select`: retorna um subconjunto das colunas de um data.frames, usando uma notação flexível;

- `pull()`: retire uma única variável;

-`filter`: extrair um subconjunto de linhas(observações) de um data.frames com base em condições lógicas;

-`arrange`: reordenar linhas de um data.frames;

-`rename`: renomear variáveis em um data.frames;

-`mutate`: adiciona novas variáveis/colunas ou transforme variáveis existentes;

-`summarise/summarize`: gera estatísticas resumidas de diferentes variáveis no data.frames, possivelmente dentro dos estratos.

4.3 Propriedades das funções do `dplyr`

As funções têm algumas características comuns:

-1.O primeiro argumento é um data.frames;

-2.Os argumentos subsequentes descrevem o que fazer com o data.frames especificado no primeiro argumento;

-3.O resultado de retorno de uma função é um novo data.frames;

-4.Os data.frames devem devidamente formatados e anotados para que tudo isso seja útil. Em particular, os dados devem estar *organizados*.

4.4 Instalando o Pacote `dplyr`

O pacote pode ser instalado a partir do CRAN ou do GitHub usando o pacote `devtools` com a função `install_github()`. O repositório GitHub normalmente contém as versões mais atualizadas dos pacotes.

Para instalar a partir do CRAN, bastar executar:

```
install.packages("dplyr")
```

Para instalar a partir do GitHub, execute:

```
library(devtools) #carregar o pacote 'devtools' antes.
devtools::install_github("hadley/dplyr")
```

Após a instalação do pacote, carregá-lo com a função `library()`:

```
library(dplyr)
```

Ao carregar o pacote você pode receber alguns avisos, porque há funções no `dplyr`que têm o mesmo nome que as funções em outros pacotes. Por enquanto pode ignorar os avisos.

4.5 `select()`

Para melhor apresentar as funcionalidades da função, usaremos um conjunto de dados diários sobre poluição do ar e taxa de mortalidade da cidade de Chicago, nos EUA.

Você pode carregar os dados no R usando a função `readRDS()`:

```
chicago <- readRDS("data/chicago.rds")
```

Este banco de dados encontra no seguinte endereço: http://www.biostat.jhsph.edu/~rpeng/leanpub/rprog/chicago_data.zip e está em um arquivo zipado. Uma das formas para facilitar o processo de descompactação do arquivo pelo R é:

```
# objeto caracter, endereço do arquivo.
fileURL <- "http://www.biostat.jhsph.edu/~rpeng/leanpub/rprog/chicago_data.zip"

#Esta função pode ser usada para baixar um arquivo da Internet.
download.file(fileURL, destfile = "data/chicago.rds", method = "curl", extra='-L')
```

Descrição do banco: tem 8 colunas e 6940 linhas. Cada linha refere-se a um dia. As colunas são:

- city:
  - cidade, neste campo tem apenas "chic" referenciando a cidade de Chicago.
- tmpd:
  - temperatura em Fahrenheit.
- dptp:
  - temperatura do ponto de orvalho.
- date:
  - tempo em dias.
- pm25tmean2:
  - partículas médias < 2,5mg por m cúbico (mais perigoso).
- pm10tmean2:
  - partículas médias em 2,5^-10 por m cúbico.
- o3tmean2:
  - Ozônio em partes por bilhão.
- no2tmean2:
  - Medição mediana de dióxido de sulfato.

Umas das formas de ter informações do seu banco de dados é utilizar as seguintes funções `dim()` e `str()`. A primeira especifica a dimensão do seu banco e a segunda a estrutura do seu banco de dados.

```
dim(chicago)
```

```
## [1] 6940    8
```

```
str(chicago)
```

```
## 'data.frame':    6940 obs. of  8 variables:
##  $ city      : chr  "chic" "chic" "chic" "chic" ...
##  $ tmpd      : num  31.5 33 33 29 32 40 34.5 29 26.5 32.5 ...
##  $ dptp      : num  31.5 29.9 27.4 28.6 28.9 ...
##  $ date      : Date, format: "1987-01-01" "1987-01-02" ...
##  $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
##  $ pm10tmean2: num  34 NA 34.2 47 NA ...
##  $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
##  $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

Muitas vezes teremos um data.frames contendo um grande número de dados. Com isso, a função `select()` permite obter as poucas colunas que você pode

precisar.

Suponhamos que desejássemos pegar as 3 primeiras colunas. Há algumas maneiras de fazer isto. Poderíamos, por exemplo, usar o índices númericos. Mas também podemos usar os nomes diretamente.

```
names(chicago[1:3])
```

```
## [1] "city" "tmpd" "dptp"
```

```
subset1 <- select(chicago, city:dptp)
head(subset1)
```

```
##   city tmpd   dptp
## 1 chic 31.5 31.500
## 2 chic 33.0 29.875
## 3 chic 33.0 27.375
## 4 chic 29.0 28.625
## 5 chic 32.0 28.875
## 6 chic 40.0 35.125
```

Normalmente `:` não pode ser usado com nomes ou sequências de caracteres, mas dentro da função `select()` pode usá-lo para especificar um intervalo de nomes de variáveis.

Pode **omitir** variáveis usando a função `select()` usando o sinal negativo.

```
subset2 <- select(chicago, -(city:dptp))
head(subset2)
```

```
##         date pm25tmean2 pm10tmean2 o3tmean2 no2tmean2
## 1 1987-01-01         NA   34.00000 4.250000  19.98810
## 2 1987-01-02         NA         NA 3.304348  23.19099
## 3 1987-01-03         NA   34.16667 3.333333  23.81548
## 4 1987-01-04         NA   47.00000 4.375000  30.43452
## 5 1987-01-05         NA         NA 4.750000  30.33333
## 6 1987-01-06         NA   48.00000 5.833333  25.77233
```

o que indica que estamos incluindo todas as variáveis, exceto as variáveis `city` até `dptp`.

O código equivalente ao anterior sem o uso do pacote seria:

```
i <- match("city", names(chicago))
j <- match("dptp", names(chicago))
head(chicago[, -(i:j)])
```

```
##         date pm25tmean2 pm10tmean2 o3tmean2 no2tmean2
## 1 1987-01-01         NA   34.00000 4.250000  19.98810
## 2 1987-01-02         NA         NA 3.304348  23.19099
## 3 1987-01-03         NA   34.16667 3.333333  23.81548
## 4 1987-01-04         NA   47.00000 4.375000  30.43452
## 5 1987-01-05         NA         NA 4.750000  30.33333
## 6 1987-01-06         NA   48.00000 5.833333  25.77233
```

A função de correspondência `mathc()` retorna um vetor das posições das (primeiras) correspondências de seu primeiro argumento no segundo. De acordo com a Documentação R, a função é equivalente ao operador `%in%` que indica se uma correspondência foi localizada para o vetor1 no vetor2. O valor do resultado será VERDADEIRO ou FALSO, mas nunca NA. Portanto, o operador `%in%` pode ser útil em condições `if`.

Exemplos:

```
#função math().
v1 <- c("a1","b2","c1","d2")
v2 <- c("g1","x2","d2","e2","f1","a1","c2","b2","a2")
x <- match(v1,v2)
x
```

```
## [1]  6  8 NA  3
```

```
#com o operador %in%.
v1 <- c("a1","b2","c1","d2")
v2 <- c("g1","x2","d2","e2","f1","a1","c2","b2","a2")
v1 %in% v2
```

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

A função `select()` permite uma sintaxe especial que especifica nomes de variáveis com base em padrões. Por exemplo, há várias funções auxiliares que você pode usar:

- 1.`starts_with("abc")`: corresponde aos nomes que começam com “abc”;

```
#Queremos manter todas as variáveis que começam com um "d":
subset3 <- select(chicago, starts_with("d"))
head(subset3)
```

```
##     dptp       date
## 1 31.500 1987-01-01
## 2 29.875 1987-01-02
## 3 27.375 1987-01-03
## 4 28.625 1987-01-04
## 5 28.875 1987-01-05
## 6 35.125 1987-01-06
```

- 2.`ends_with("xyz")`: corresponde aos nomes que terminam com “xyz”;

```
subset4 <- select(chicago, ends_with("2"))
head(subset4)
```

```
##   pm25tmean2 pm10tmean2 o3tmean2 no2tmean2
## 1         NA   34.00000 4.250000  19.98810
## 2         NA         NA 3.304348  23.19099
## 3         NA   34.16667 3.333333  23.81548
## 4         NA   47.00000 4.375000  30.43452
## 5         NA         NA 4.750000  30.33333
## 6         NA   48.00000 5.833333  25.77233
```

- 3.`contains("ijk")`: corresponde aos nomes que contêm “ijk”;

```
subset5 <- select(chicago, contains("tmean"))
head(subset5)
```

```
##   pm25tmean2 pm10tmean2 o3tmean2 no2tmean2
## 1         NA   34.00000 4.250000  19.98810
## 2         NA         NA 3.304348  23.19099
## 3         NA   34.16667 3.333333  23.81548
## 4         NA   47.00000 4.375000  30.43452
## 5         NA         NA 4.750000  30.33333
## 6         NA   48.00000 5.833333  25.77233
```

- 4.`matches("(.)\\1")`: selecionar variáveis que correspondem a uma expressão regular. Esta corresponde a qualquer variável que contenha caracteres repetidos. Você aprenderá mais sobre expressões regulares no capítulo Strings do livro R for data science.

```
subset6 <- select(chicago, matches(c(".m."), names(chicago)))
head(subset6)
```

```
##   tmpd pm25tmean2 pm10tmean2 o3tmean2 no2tmean2
## 1 31.5         NA   34.00000 4.250000  19.98810
## 2 33.0         NA         NA 3.304348  23.19099
## 3 33.0         NA   34.16667 3.333333  23.81548
## 4 29.0         NA   47.00000 4.375000  30.43452
## 5 32.0         NA         NA 4.750000  30.33333
## 6 40.0         NA   48.00000 5.833333  25.77233
```

- 5.`num_range("x", 1:3)`: Corresponde x1, x2 e x3.

```
#Criando um objeto df que é um data frame
df <- as.data.frame(matrix(runif(100), nrow = 10))
df <- tbl_df(df[c(3, 4, 7, 1, 9, 8, 5, 2, 6, 10)])
select(df, V4:V6)
```

```
## # A tibble: 10 x 8
##        V4     V7     V1    V9    V8    V5    V2     V6
##     <dbl>  <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>
##  1 0.815  0.313  0.744  0.355 0.809 0.177 0.487 0.341
##  2 0.413  0.950  0.865  0.686 0.117 0.289 0.468 0.463
##  3 0.723  0.275  0.0296 0.728 0.394 0.524 0.948 0.747
##  4 0.864  0.0351 0.590  0.853 0.480 0.455 0.101 0.0393
##  5 0.934  0.501  0.566  0.326 0.581 0.900 0.459 0.471
##  6 0.427  0.660  0.873  0.364 0.270 0.432 0.607 0.802
##  7 0.0498 0.733  0.792  0.999 0.981 0.970 0.230 0.733
##  8 0.618  0.957  0.135  0.248 0.248 0.642 0.776 0.260
##  9 0.485  0.0198 0.687  0.689 0.537 0.944 0.711 0.311
## 10 0.718  0.357  0.0559 0.240 0.450 0.490 0.383 0.162
```

```
select(df, num_range("V", 4:6))
```

```
## # A tibble: 10 x 3
##        V4    V5     V6
##     <dbl> <dbl>  <dbl>
```

```
##  1 0.815  0.177 0.341
##  2 0.413  0.289 0.463
##  3 0.723  0.524 0.747
##  4 0.864  0.455 0.0393
##  5 0.934  0.900 0.471
##  6 0.427  0.432 0.802
##  7 0.0498 0.970 0.733
##  8 0.618  0.642 0.260
##  9 0.485  0.944 0.311
## 10 0.718  0.490 0.162
```

Você também pode usar expressões regulares mais gerais, se necessário. Veja a página de ajuda (?select) para mais detalhes.

`select()` pode ser usado para renomear variáveis, mas raramente é útil porque descarta todas as variáveis não mencionadas explicitamente. Em vez disso, use `rename()`, que é uma variante de `select()` que mantém todas as variáveis que não são mencionadas explicitamente.

Outra opção é usar `select()` em conjunto com o `everything()` auxiliar. Isso é útil se você tiver um punhado de variáveis que deseja mover para o início do quadro de dados.

```
subset7 <- select(chicago, o3tmean2, no2tmean2, everything())
head(subset7)
```

```
##   o3tmean2 no2tmean2 city tmpd   dptp       date pm25tmean2 pm10tmean2
## 1 4.250000  19.98810 chic 31.5 31.500 1987-01-01         NA   34.00000
## 2 3.304348  23.19099 chic 33.0 29.875 1987-01-02         NA         NA
## 3 3.333333  23.81548 chic 33.0 27.375 1987-01-03         NA   34.16667
## 4 4.375000  30.43452 chic 29.0 28.625 1987-01-04         NA   47.00000
## 5 4.750000  30.33333 chic 32.0 28.875 1987-01-05         NA         NA
## 6 5.833333  25.77233 chic 40.0 35.125 1987-01-06         NA   48.00000
```

4.6 rename()

Para renomear variáveis em uma data.frames em R não é tão prático. E a função `rename()` foi projetada para facilitar esse processo.

Os nomes das cinco primeiras variáveis do data frame chicago.

```
#Imprimir às 3 primeiras linhas da primeira a quinta coluna.
head(chicago[, 1:5], 3)
```

```
##   city tmpd   dptp       date pm25tmean2
## 1 chic 31.5 31.500 1987-01-01         NA
## 2 chic 33.0 29.875 1987-01-02         NA
## 3 chic 33.0 27.375 1987-01-03         NA
```

A coluna `dptp` deve representar a temperatura do ponto de orvalho e a coluna `pm25tmean2` fornece os dados do PM2.5. No entanto, esses nomes são bastante obscuros ou estranhos e provavelmente serão renomeados para algo mais sensato.

```
chicago <- rename(chicago, Temp_Orv = dptp, pm25 = pm25tmean2)
head(chicago[, 1:5], 3)
```

```
##   city tmpd Temp_Orv       date pm25
## 1 chic 31.5   31.500 1987-01-01   NA
## 2 chic 33.0   29.875 1987-01-02   NA
## 3 chic 33.0   27.375 1987-01-03   NA
```

A sintaxe dentro da `rename()` função é ter o novo nome no lado esquerdo do `=` sinal e o nome antigo no lado direito.

4.6.1 Exercícios

4.7 `mutate()`

Em certas situações é útil adicionar novas colunas/variáveis que são funções de colunas existentes no data frames, ou seja, criar novas variáveis derivadas de variáveis existentes. Esse é o trabalho de `mutate()`. Esta função adiciona novas colunas no final do seu conjunto de dados. `mutate()` fornece uma interface limpa para fazer isso. Lembre-se de que, quando você está no RStudio, a maneira mais fácil de ver todas as colunas é `View()`.

Por exemplo, com os dados de poluição do ar, subtraindo a média dos dados. Dessa forma, podemos verificar se o nível de poluição do ar de um determinado dia é maior ou menor que a média (em oposição a observar seu nível absoluto).

Aqui, criamos uma variável `pm25difmean` que subtrai a média da variável `pm25`.

```
chicago <- mutate(chicago, pm25difmean = pm25 - mean(pm25, na.rm = TRUE))
head(chicago)
```

```
##   city tmpd Temp_Orv       date pm25 pm10tmean2 o3tmean2 no2tmean2
## 1 chic 31.5   31.500 1987-01-01   NA   34.00000 4.250000  19.98810
## 2 chic 33.0   29.875 1987-01-02   NA         NA 3.304348  23.19099
## 3 chic 33.0   27.375 1987-01-03   NA   34.16667 3.333333  23.81548
## 4 chic 29.0   28.625 1987-01-04   NA   47.00000 4.375000  30.43452
## 5 chic 32.0   28.875 1987-01-05   NA         NA 4.750000  30.33333
## 6 chic 40.0   35.125 1987-01-06   NA   48.00000 5.833333  25.77233
##   pm25difmean
## 1          NA
## 2          NA
## 3          NA
## 4          NA
## 5          NA
## 6          NA
```

Há também a função relacionada `transmute()`, que faz a mesma coisa que, `mutate()`, mas elimina todas as variáveis não transformadas.

Aqui, desprezamos as variáveis PM10 e ozônio (O3).

```
chicago %>%
  transmute(pm10difmean = pm10tmean2 - mean(pm10tmean2, na.rm = TRUE),
```

```
              O3difmean = o3tmean2 - mean(o3tmean2, na.rm = TRUE)) %>%
  head()
```

```
##   pm10difmean O3difmean
## 1   0.1047939 -15.18551
## 2          NA -16.13117
## 3   0.2714605 -16.10218
## 4  13.1047939 -15.06051
## 5          NA -14.68551
## 6  14.1047939 -13.60218
```

Observe que existem apenas duas colunas no quadro de dados transformados.

Há inúmeras funções que pode ser feita, a propriedade é que a função deva ser vetorizada: ela deve pegar um vetor de valores como entrada, retornar um vetor com o mesmo número de valores que a saída.

4.8 arrange()

A função `arrange()` é usada para reordenar linhas de um quadro de dados de acordo com uma das variáveis/colunas. Reordenar linhas de um quadro de dados (preservando a ordem correspondente de outras colunas) normalmente é uma tarefa difícil em R. A função simplifica bastante o processo.

Aqui, podemos ordenar as linhas do quadro de dados por data, para que a primeira linha seja a observação mais antiga e a última linha seja a observação mais recente.

```
chicago %>%
  arrange(date) %>%
  head()
```

```
##   city tmpd Temp_Orv       date pm25 pm10tmean2 o3tmean2 no2tmean2
## 1 chic 31.5   31.500 1987-01-01   NA   34.00000 4.250000  19.98810
## 2 chic 33.0   29.875 1987-01-02   NA         NA 3.304348  23.19099
## 3 chic 33.0   27.375 1987-01-03   NA   34.16667 3.333333  23.81548
## 4 chic 29.0   28.625 1987-01-04   NA   47.00000 4.375000  30.43452
## 5 chic 32.0   28.875 1987-01-05   NA         NA 4.750000  30.33333
## 6 chic 40.0   35.125 1987-01-06   NA   48.00000 5.833333  25.77233
##   pm25difmean
## 1          NA
## 2          NA
## 3          NA
## 4          NA
## 5          NA
## 6          NA
```

Agora podemos verificar as primeiras linhas:

```
chicago %>%
  select(date, pm25) %>%
  head(3)
```

```
##         date pm25
## 1 1987-01-01   NA
## 2 1987-01-02   NA
## 3 1987-01-03   NA
```

e as últimas linhas:

```
chicago %>%
  select(date, pm25) %>%
  tail()
```

```
##            date     pm25
## 6935 2005-12-26  8.40000
## 6936 2005-12-27 23.56000
## 6937 2005-12-28 17.75000
## 6938 2005-12-29  7.45000
## 6939 2005-12-30 15.05714
## 6940 2005-12-31 15.00000
```

As colunas também podem ser organizadas em ordem decrescente, usando o operador especial `desc()`.

```
chicago <- arrange(chicago, desc(date))
```

Observa as três primeiras e as últimas três linhas mostra as datas em ordem decrescente.

```
chicago %>%
  select(date, pm25) %>%
  head()
```

```
##         date     pm25
## 1 2005-12-31 15.00000
## 2 2005-12-30 15.05714
## 3 2005-12-29  7.45000
## 4 2005-12-28 17.75000
## 5 2005-12-27 23.56000
## 6 2005-12-26  8.40000
```

```
chicago %>%
  select(date, pm25) %>%
  tail()
```

```
##            date pm25
## 6935 1987-01-06   NA
## 6936 1987-01-05   NA
## 6937 1987-01-04   NA
## 6938 1987-01-03   NA
## 6939 1987-01-02   NA
## 6940 1987-01-01   NA
```

4.9 filter()

A função `filter()` é usada para extrair subconjuntos de linhas de um data frame. O primeiro argumento é o nome do quadro de dados. O segundo argumento e os argumentos subseqüentes são as expressões que filtram o quadro de dados.

Suponhamos que desejássemos extrair as linhas do banco `chicago` em que o níveis de PM2,5 sejam maiores que 30, poderíamos fazer

```
chicago <- as_tibble(chicago)
chicago %>%
  filter(pm25 > 30) %>%
  head()
```

```
## # A tibble: 6 x 9
##   city   tmpd Temp_Orv date        pm25 pm10tmean2 o3tmean2 no2tmean2
##   <chr> <dbl>    <dbl> <date>     <dbl>      <dbl>    <dbl>     <dbl>
## 1 chic     37     35.2 2005-12-24  30.8       25.2     1.77      32.0
## 2 chic     41     32.6 2005-12-23  32.9       34.5     6.91      29.1
## 3 chic     22     23.3 2005-12-22  36.6       42.5     5.39      33.7
## 4 chic     12      7.7 2005-12-21  37.9       59.5     3.66      34.9
## 5 chic      8     -1.8 2005-12-07  37.8       39       3.92      34.3
## 6 chic     55     49.8 2005-11-08  40         36.5     4.10      27.2
## # ... with 1 more variable: pm25difmean <dbl>
```

Quando você executa essa linha de código, o dplyr executa a operação de filtragem e retorna um novo quadro de dados. As funções `dplyr` nunca modificam suas entradas; portanto, se você deseja salvar o resultado, precisará usar o operador de atribuição `<-`.

Para usar a filtragem de maneira eficaz, você precisa saber como selecionar as observações que deseja usando os operadores de comparação. R fornece o conjunto padrão: `>`, `>=`, `<`, `<=`, `!=`(não igual), e `==`(igual).

Quando você começa com R, o erro mais fácil de cometer é usar `=` em vez de `==` testar a igualdade. Quando isso acontece, você recebe um erro informativo:

```
filter(chicago, tmpd = 33)
#> Error: `month` (`month = 1`) must not be named, do you need `==`?
```

Há outro problema comum que você pode encontrar ao usar `==`: números de ponto flutuante. Esses resultados podem surpreendê-lo!

```
sqrt(2) ^ 2 == 2
```

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

```
1 / 49 * 49 == 1
```

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

Os computadores usam aritmética de precisão finita (eles obviamente não podem armazenar um número infinito de dígitos!). Lembre-se de que todo número que você vê é uma aproximação. Use `near()`:

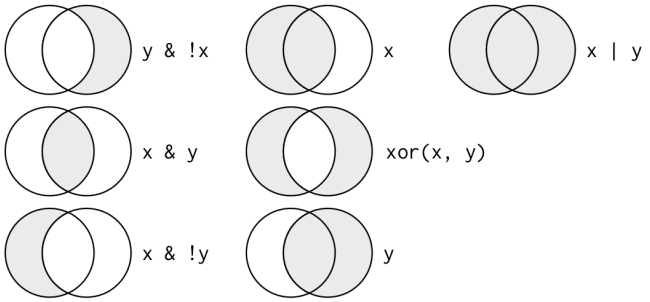


Figura 4.1: Conjunto completo de operações booleanas. x é o círculo do lado esquerdo, y é o círculo do lado direito e a região sombreada mostra quais partes cada operador seleciona.

```
near(sqrt(2) ^ 2,  2)
```

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

```
near(1 / 49 * 49, 1)
```

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

4.9.1 Operadores lógicos

Vários argumentos para `filter()` são combinados com "e": toda expressão deve ser verdadeira para que uma linha seja incluída na saída. Para outros tipos de combinações, você precisará usar operadores booleanos: `&` é "e", `|` é "ou" e `!` é "negação". A Figura abaixo mostra o conjunto completo de operações booleanas.

O código a seguir localiza todas as temperaturas iguais a 30ºF ou 40ºF no banco:

```
chicago %>%
  filter(tmpd == 30 | tmpd == 40)
```

```
## # A tibble: 136 x 9
##    city   tmpd Temp_Orv date        pm25 pm10tmean2 o3tmean2 no2tmean2
##    <chr> <dbl>    <dbl> <date>     <dbl>      <dbl>    <dbl>     <dbl>
##  1 chic     40     33.6 2005-12-27  23.6       27       4.47      23.5
##  2 chic     30     27.9 2005-12-15  14.4       16.5     4.90      25.4
##  3 chic     30     21.1 2005-11-18  11.7       22.4     4.53      25.0
##  4 chic     40     36.7 2005-10-23   8.5       10      11.6       18.1
##  5 chic     40     25.1 2005-05-02   8         24      17.0       15.4
##  6 chic     30     22.7 2005-03-11  12.0       20.5    23.1       19.9
##  7 chic     30     18.2 2005-02-16  12.8       27.5    19.1       19.9
##  8 chic     30     25   2005-02-03  47.4       53.6     7.27      48.6
##  9 chic     30     26.2 2005-01-30  NA         23      12.4       23.4
## 10 chic     30     27.6 2005-01-09  19.1       17      11.4       19.0
## # ... with 126 more rows, and 1 more variable: pm25difmean <dbl>
```

Às vezes, você pode simplificar um subconjunto complicado lembrando a lei de De Morgan: `!(x & y)` é o mesmo que `!x | !y` e `!(x | y)` é o mesmo que `!x`

`& !y`. Por exemplo, se você deseja encontrar temperaturas voos que não foram atrasados (na chegada ou na partida) por mais de duas horas, você pode usar um dos dois filtros a seguir:

```
chicago %>%
  filter(!(tmpd > 30 & pm25 < 15))
```

```
## # A tibble: 2,156 x 9
##    city   tmpd Temp_Orv date        pm25 pm10tmean2 o3tmean2 no2tmean2
##    <chr> <dbl>    <dbl> <date>     <dbl>      <dbl>    <dbl>      <dbl>
##  1 chic     35     30.1 2005-12-31  15         23.5     2.53       13.2
##  2 chic     36     31   2005-12-30  15.1       19.2     3.03       22.8
##  3 chic     37     34.5 2005-12-28  17.8       27.5     3.26       19.3
##  4 chic     40     33.6 2005-12-27  23.6       27       4.47       23.5
##  5 chic     37     35.2 2005-12-24  30.8       25.2     1.77       32.0
##  6 chic     41     32.6 2005-12-23  32.9       34.5     6.91       29.1
##  7 chic     22     23.3 2005-12-22  36.6       42.5     5.39       33.7
##  8 chic     12      7.7 2005-12-21  37.9       59.5     3.66       34.9
##  9 chic     13      7.7 2005-12-20  25.8       32       3.85       32.9
## 10 chic      5     -0.3 2005-12-19  21.2       21       8.06       31.8
## # ... with 2,146 more rows, and 1 more variable: pm25difmean <dbl>
```

4.10 summarise()

O último verbo-chave é `summarise()`. Recolhe um quadro de dados em uma única linha:

Capítulo 5

Vôos em Nova York de 2013

Neste capítulo, vamos nos concentrar em como usar o pacote `dplyr` e o que aprendemos no capítulo anterior. Ilustraremos as ideias principais usando dados do pacote `nycflights13` e usaremos o `ggplot2` para nos ajudar a entender os dados.

```
library(nycflights13)
library(tidyverse) # ou isoladamente: library(dplyr).
```

5.1 nycflights13

Este quadro de dados contém todos os 336.776 vôos que partiram de Nova York em 2013. Os dados são do Bureau of Transportation Statistics dos EUA e estão documentados em `?flights`.

```
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Para ver todo o conjunto de dados, você pode executar o `View(flights)` que abrirá o conjunto de dados no visualizador do RStudio.

As abreviações de letras sob os nomes das colunas descrevem o tipo de cada variável:

`-int` significa números inteiros;

`-dbl` significa números duplos ou reais;

`-chr` significa vetores de caracteres ou seqüências de caracteres;

`-dttm` significa data e hora (uma data + uma hora).

`-lgl` significa vetores lógicos que contêm apenas `TRUE` ou `FALSE`;

`-fctr` significa fatores, que R usa para representar variáveis categóricas com valores possíveis fixos.

`-data` significa data.

Existem outros tipos comuns de variáveis que não são usadas neste conjunto de dados.

5.2 Formato

Vamos entender o formato do nosso banco e suas variáveis. Fazendo alterações e modificações necessárias para melhor compreender o processo.

Colunas do quadro de dados:

- year, month e day referência a data de partida. Poderemos alterar os nomes para o nosso vernáculo.
- dep_time e arr_time: horários reais de partida e chegada (formato em minutos)
- sched_dep_time, sched_arr_time: horários de partida e chegada programados (formato em minutos)
- dep_delay, arr_delay: Atrasos de partida e chegada, em minutos. Tempos negativos representam partidas/chegadas antecipadas.
- carrier: códigos de operadoras das companhias aéreas.
- flight: número do vôo.
- tailnum: número da cauda do avião.
- origin, dest: origem e destino.
- air_time: quantidade de tempo gasto no ar, em minutos.
- distance: distância entre aeroportos, em milhas.
- hour, minute: hora da partida programada dividida em hora e minutos.
- time_hour: data e hora agendadas do voo como uma data POSIXct. Juntamente com a origem, pode ser usado para unir dados de voos a dados meteorológicos.

```
Voos <- flights %>%
  rename(Ano = year, "Mês" = month, Dia = day, )
```

Capítulo 6

Conclusão

...

Peng, R.D. and Welty, L.J. (2004) The NMMAPSdata package. R News 4(2).

Wood, S.N. (2006, 2017) Generalized Additive Models: An Introduction with R.

Peng, R.D., Exploratory Data Analysis with R (2016) This version was published on 2016-07-20. This book is for sale at http://leanpub.com/exdata

OLIVEIRA, Paulo Felipe de; GUERRA, Saulo; MCDONNELL, Robert. Ciência de Dados com R – Introdução. Brasília: Editora IBPAD, 2018.

https://www.curso-r.com/material/pipe/

https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html

Dasu T, Johnson T (2003). Exploratory Data Mining and Data Cleaning. Wiley-IEEE.

http://leg.ufpr.br/~fernandomayer/aulas/ce083-2016-2/02_funcoes_e_objetos.html#classes_de_objetos

Apêndice A: Respostas dos Exercícios