



**Cursos Integrados
em Vigilância em Saúde**

Curso

**Análise de dados para a vigilância
em saúde – Curso Básico**

Módulo 3 - Gerenciamento de dados na vigilância em saúde

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Reitor Irineu Manoel de Souza

Vice-Reitora Joana Célia dos Passos

Pró-Reitora de Pós-graduação Werner Kraus

Pró-Reitor de Pesquisa e Inovação Jacques Mick

Pró-Reitor de Extensão Olga Regina Zigelli Garcia

CENTRO DE CIÊNCIAS DA SAÚDE

Diretor Fabrício de Souza Neves

Vice-Diretora Ricardo de Souza Magini

DEPARTAMENTO DE SAÚDE PÚBLICA

Chefe do Departamento Rodrigo Otávio Moretti Pires

Subchefe do Departamento Sheila Rúbia Lindner

Coordenadora do Curso Alexandra Crispim Boing

INSTITUTO TODOS PELA SAÚDE (ITPS)

Diretor presidente Jorge Kalil (Professor titular da Faculdade de Medicina da Universidade de São Paulo; Diretor do Laboratório de Imunologia do Incor)

ASSOCIAÇÃO BRASILEIRA DE SAÚDE COLETIVA (ABRASCO)

Presidente Rosana Teresa Onocko Campos

EQUIPE DE PRODUÇÃO

Denis de Oliveira Rodrigues

Kamila de Oliveira Belo

Marcelo Eduardo Borges

Oswaldo Gonçalves Cruz

Alexandra Crispim Boing

Antonio Fernando Boing

Módulo 3 - Gerenciamento de dados na vigilância em saúde

Curso _____

**Análise de dados para a vigilância
em saúde – Curso Básico**

Dados Internacionais de Catalogação-na-Publicação (CIP)

G367 Gerenciamento de dados na vigilância em saúde/ Kamila de Oliveira Belo, Denis de Oliveira Rodrigues, Oswaldo Gonçalves Cruz, Marcelo Eduardo Borges . – Santa Catarina ; São Paulo ; Rio de Janeiro : UFSC ; ITPS ; Abrasco; 2022. 121p. (Análise de dados para a vigilância em saúde – Curso Básico; Módulo 3).

Publicação Online
10.52582/curso-analise-dados-vigilancia-modulo3

1. Vigilância em saúde 2. Gerenciamento de dados I. Título

Sumário

| | |
|--|-----------|
| Manipulação de dados na vigilância em saúde | 06 |
| 1. Obtendo os dados para sua análise..... | 10 |
| 2. Limpando e transformando os dados de vigilância | 11 |
| 3. 3. Avaliando a consistência dos dados | 14 |
| 3.1 Selecionando as variáveis que serão analisadas..... | 18 |
| 3.2 Simplificando o fluxo de análises | 19 |
| 4. 4. Preparando as variáveis para análise de dados de vigilância | 22 |
| 4.1 Excluindo colunas na tabela | 22 |
| 4.2 Inserindo novas colunas à tabela | 24 |
| 4.3 Limpando caracteres e padronizando colunas | 27 |
| 4.4 Filtrando colunas | 30 |
| 4.5 Unindo colunas | 34 |
| 4.6 Resumindo os valores de uma coluna | 38 |
| 4.7 Modificando a ordem das colunas..... | 40 |
| 5. Avaliando a completude dos dados..... | 42 |
| 5.1 Renomeando valores | 46 |
| 6. Arrumando os dados de vigilância em saúde | 49 |
| 6.1 Organizando dados em linhas e colunas (dados retangulares)..... | 50 |
| 6.2 Modificando o formato de um banco de dados | 52 |
| 6.3 Separar conteúdo de variáveis em mais colunas..... | 59 |
| 6.4 Recodificando linhas e colunas..... | 64 |

| | |
|---|-----|
| 7. Unindo tabelas com o R | 71 |
| 7.1 Tipos de cruzamento de dados (Join) | 73 |
| 8. Transformando e limpando textos | 77 |
| 8.1 Contando caracteres e extraindo textos | 78 |
| 8.2 Alterando a ordem e reordenando texto | 82 |
| 8.3 Unindo textos | 84 |
| 8.4 Transformando textos | 86 |
| 8.5 Identificando padrões | 90 |
| 8.6 Substituindo valores de texto | 95 |
| 9. Organizando os eventos de vigilância no tempo | 97 |
| 9.1 Transformando datas | 97 |
| 9.2 Cálculo com datas | 101 |
| 9.3 Extrair dia, mês e ano, dia da semana | 108 |
| 9.4 Identificar semana epidemiológica das datas | 110 |
| 9.5 Sequência de datas | 117 |
| 10. Exportação dos dados | 119 |
| 10.1 Exportação para a extensão CSV | 119 |
| 10.2 Exportação para a extensão XLSX | 120 |

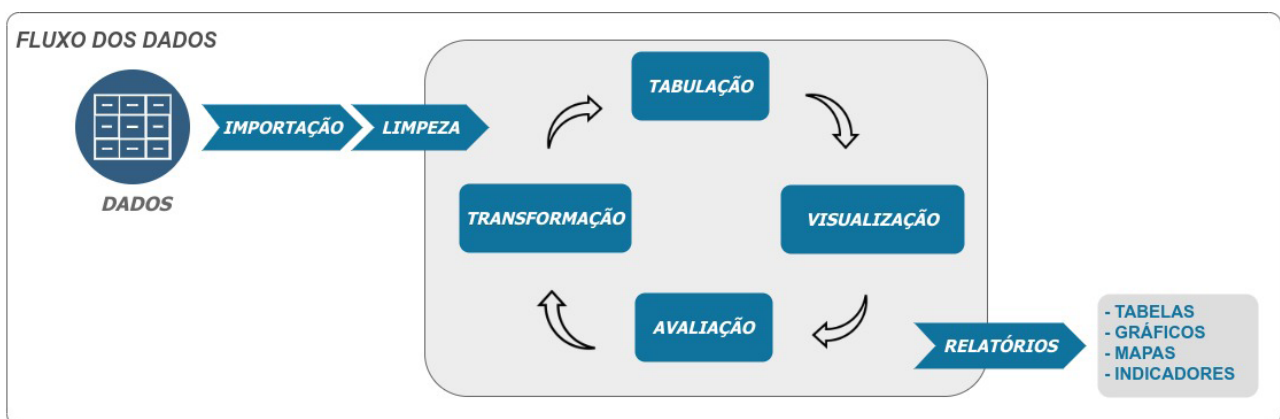
Manipulação de dados na vigilância em saúde

Em seu dia a dia, por vezes o profissional de vigilância é convocado a produzir “informação para a ação”, o que torna crucial uma resposta rápida e eficiente às demandas de saúde. O [Guia de Vigilância Epidemiológica](#) em sua 7ª edição lista enquanto principais funções da vigilância epidemiológica:

- coleta de dados;
- processamento de dados coletados;
- análise e interpretação dos dados processados;
- recomendação das medidas de prevenção e controle apropriadas;
- promoção das ações de prevenção e controle indicadas;
- avaliação da eficácia e efetividade das medidas adotadas;
- divulgação das informações pertinentes.

Observe na Figura 1 as etapas que compõem a manipulação de dados de vigilância:

Figura 1: Ciclo de manipulação dos dados de vigilância em saúde.



Neste módulo você irá participar e intervir junto a todo o ciclo de produção de informação de vigilância, desde a coleta dos dados até a produção de relatórios que subsidiem as ações de saúde.

Ao final do Módulo 3 você será capaz de realizar uma rotina sistemática para as seguintes etapas de análise dos dados de vigilância em saúde:

1. Limpar e transformar seus dados.
2. Criar tabelas com filtros escolhidos.
3. Unir/linkar dados de diversas fontes.
4. Exportar os dados tratados para diferentes formatos.

Para seguir com este módulo do curso você deve já ter instalado o *software* R e a interface gráfica RStudio e conhecer os conceitos básicos para análise de dados na vigilância em saúde. Esses passos estão disponíveis nos Módulos 1 e 2 deste curso!

Nesse momento instale e carregue todos os pacotes aqui listados para você possa utilizá-los ao longo deste módulo:

- `foreign`
- `readxl`
- `readr`
- `janitor`
- `skimr`
- `stringr`
- `stringi`
- `lubridate`
- `summarytools`
- `descr`

Vamos lá, para isso inicie o seu RStudio, crie um novo *script* e replique o código abaixo:

```
if(!require(foreign)) install.packages("foreign");library(foreign)
if(!require(readxl)) install.packages("readxl");library(readxl)
if(!require(readr)) install.packages("readr");library(readr)
if(!require(janitor)) install.packages("janitor");library(janitor)
if(!require(skimr)) install.packages("skimr");library(skimr)
if(!require(stringr)) install.packages("stringr");library(stringr)
if(!require(stringi)) install.packages("stringi");library(stringi)
if(!require(lubridate)) install.packages("lubridate");library(lubridate)
```

Com a execução do código acima, você está carregando os pacotes e instalando-os em sua máquina. Ao longo deste módulo, utilizaremos funções vinculadas aos pacotes instalados e aprofundaremos seus conceitos e usos.

Não será necessário instalar estes pacotes novamente em futuras análises e rotinas da vigilância. Para utilizá-los basta executar a função `require()` ou `library()` que carregarão um pacote já instalado. Veja como ficariam os códigos:



```
# Utilizando a função library para carregar um pacote já instalado
library("readr")

# Utilizando a função require para carregar um pacote já instalado
require("readr")
```

Durante este curso utilizaremos sempre o uso da função `library()` para instalar e carregar pacotes!

1. Obtendo os dados para sua análise

Neste módulo vamos simular um problema fictício que ocorrerá no Estado de Rosas. Vamos lá! Você enquanto membro da Vigilância Epidemiológica estadual de Rosas recebeu um alerta do Ministério da Saúde indicando aumento de casos de hepatites virais em pessoas de até 17 anos em estados brasileiros.

Será necessário construir uma análise da situação de saúde para o alerta apresentado. Para esta análise, você precisará manipular diversas bases de dados que subsidiem a construção de um relatório sobre a situação das hepatites virais em Rosas.

Vamos iniciar nossa avaliação escolhendo o banco de dados apropriado para encontrar os casos de hepatites virais notificados: o `{NINDINET.dbf}`, resultado do preenchimento da Ficha Individual de Notificação (FIN), exportada por meio do sistema **Sinan Net** - Sistema de Informação de Agravos de Notificação. Esta ficha é preenchida quando há suspeita ou confirmação da ocorrência de problema de saúde de notificação compulsória de interesse nacional, estadual ou municipal.

O primeiro passo para iniciar a sua análise de dados é a importação dos bancos de dados escolhidos para o ambiente do **RStudio**. Você aprendeu o passo a passo de como importar dados exportados dos sistemas de informação em saúde do tipo `.dbf`, arquivos do Microsoft Excel (`.xls` e `.xlsx`) e `.csv` no Módulo 2 deste curso.

Observe e replique os comandos do *script* abaixo em seu ``RStudio` e importe o banco de dados `{NINDINET.dbf}`, disponível no Ambiente Virtual do curso, para análise:

```
# Carregando o pacote foreign no RStudio
library(foreign)

# Importando o arquivo {NINDINET.dbf} no objeto {base}
# Utilizando o argumento "as.is = TRUE" para transformar os dados em caracteres
base <- read.dbf(file = 'Dados/NINDINET.dbf', as.is = TRUE)
```



Lembre-se de antes de iniciar o módulo, acessar o arquivo `Modulo_3.Rproj` disponível no menu lateral “Arquivos” do módulo para iniciar o projeto do curso. Esta etapa está descrita no Módulo 1.

Ao clicar no arquivo `Modulo_3.Rproj`, sua sessão no `RStudio` será aberta e você indicará o diretório de trabalho em que sua sessão esta para o `R`, facilitando o acesso às pastas e arquivos indicando o caminho correto de forma automatizada em qualquer computador.

Observe neste módulo que utilizaremos funções específicas de acordo com o tipo de arquivo e as suas particularidades. Vamos em frente!

2. Limpando e transformando os dados de vigilância

O sistema de vigilância no Brasil nos impõe desafios desde a coleta até o processamento dos dados nos diversos níveis da federação envolvidos. Para realizarmos a análise de situação de saúde para o Estado de Rosas iremos avaliar alguns aspectos da qualidade dos dados que serão analisados. Neste tópico o profissional de vigilância poderá avaliar consistência, completitude e validade dos dados que serão analisados.

É importante saber que para a realização dos cálculos, modelagem estatística e visualização de dados, o `R` oferece uma série de pacotes com um conjunto de funções que facilitam e otimizam estas tarefas, tornando-as muito mais fluidas e simplificadas. Atualmente, alguns pacotes integram uma série de ferramentas nas quais há uma filosofia de design, gramática e estruturas de dados em comum para serem utilizados em conjunto. Esse universo é chamado de `tidyverse`, um metapacote projetado com a finalidade de apoiar as análises na área de ciência de dados.

Neste curso iremos utilizar o **tidyverse** para criação de rotina de limpeza, organização e transformação dos seus dados de vigilância em saúde. Para isto, o profissional de vigilância deve saber que ao carregar o pacote **tidyverse**, ele carregará simultaneamente oito pacotes do R. São eles:

| Pacote | Descrição |
|----------------|---|
| ggplot2 | Cria gráficos baseado na gramática dos gráficos. |
| tibble | Leitura e gravação de um moderno formato de banco de dados. |
| tidyr | Auxílio para arrumar e transformar banco de dados. |
| readr | Leitura e gravação de arquivos delimitados por vírgulas ou tabulação. |
| purrr | Otimização e desempenho de algoritmos. |
| dplyr | Manipulação e limpeza de dados. |
| stringr | Manipulação de texto. |
| forcats | Manipulação de fatores. |

Nesta parte do módulo, iremos avaliar se o pacote **tidyverse** se encontra instalado. Caso não esteja, faremos a sua instalação. Rode o código abaixo e analise as mensagens que serão retornadas em sua tela:

```
if(!require(tidyverse)) install.packages("tidyverse")
```

```
#> Carregando pacotes exigidos: tidyverse
```

```
#> — Attaching packages —  
— tidyverse 1.3.2 —  
#> SOH ggplot2 3.3.6      SOH purrr 0.3.4  
#> SOH tibble 3.1.8      SOH dplyr 1.0.9  
#> SOH tidyr 1.2.0        SOH forcats 0.5.2  
#> — Conflicts —  
tidyverse_conflicts() —  
#> SOH lubridate::as.difftime() masks base::as.difftime()  
#> SOH lubridate::date()        masks base::date()  
#> SOH dplyr::filter()          masks stats::filter()  
#> SOH lubridate::intersect()   masks base::intersect()  
#> SOH dplyr::lag()             masks stats::lag()  
#> SOH lubridate::setdiff()     masks base::setdiff()  
#> SOH lubridate::union()       masks base::union()
```

Perceba que os pacotes são carregados em conjunto. Porém, há o retorno na tela de conflitos que ocorrem com funções que têm o mesmo nome em outro pacote já carregado no ambiente do R.

Caso o pacote já esteja instalado, é possível carregar o **tidyverse** utilizando apenas o seguinte comando:

```
library(tidyverse)
```



Atenção

A mensagem que retornou informa que as funções de mesmo nome carregadas previamente são substituídas por aquelas carregadas depois, ou seja, caso necessite utilizar as funções **filter()** do pacote **stats** deverá identificá-la como **stats::filter()**. Isso pode ser um incômodo quando há muitos pacotes carregados.

3. Avaliando a consistência dos dados

A avaliação de consistência dos dados é uma etapa rotineira e essencial do profissional de vigilância em saúde. Nela, os dados são checados para verificação de sua coerência. Esta análise de consistência permite a melhoria no rastreamento e monitoramento das doenças, agravos e outros eventos em saúde pública.

Nesta etapa você permanecerá utilizando o pacote `dplyr` que está contido no universo do metapacote `tidyverse`. O pacote `dplyr` oferece muitos recursos para transformação de bases de dados, seja ela necessária nas colunas (variáveis) ou nas linhas (registros). No quadro abaixo listamos as funções mais utilizadas no pacote `dplyr`:

| Função | Descrição |
|--------------------------|---|
| <code>select()</code> | Selecionar, excluir e organizar as colunas do banco de dados. |
| <code>mutate()</code> | Cria colunas e transforma as existentes. |
| <code>summarise()</code> | Cria colunas utilizando operações de síntese das linhas. |
| <code>group_by()</code> | Cria grupos de registros baseados em colunas. |
| <code>filter()</code> | Filtra um conjunto de registros baseado em critérios de inclusão ou exclusão. |
| <code>arrange()</code> | Ordena os valores dos registros conforme colunas definidas. |
| <code>slice()</code> | Seleciona linhas específicas. |
| <code>glimpse()</code> | Fornece um sumário dos seus dados. |



Atenção

O pacote `dpLyr` oferece muitos recursos para transformação de bases de dados e suas funções nos remetem a ações comuns, os chamados 'verbos'.

Lembre-se que o profissional de vigilância do Estado de Rosas necessita avaliar os casos de hepatites virais e, para isso ele precisará examinar os dados e escolher as variáveis adequadas para análise, corrigindo distorções ou informações inadequadas.

O primeiro passo para avaliar a consistência dos dados é importar o banco de dados do Sinan Net `{NINDINET.dbf}` disponível no menu lateral "Arquivos", Ambiente Virtual do curso. Aqui chamaremos o objeto da análise de `{base}`. Observe os comandos abaixo e replique-os em seu `RStudio`:

```
# Criando o objeto {'base'} que armazenará o banco de dados {'NINDINET.dbf'}  
base <- read.dbf(file = 'Dados/NINDINET.dbf', as.is = TRUE)
```

Digitando a função `glimpse()` você conseguirá visualizar a área inicial do banco de dados. Com este comando será possível visualizar o número de linhas da sua tabela (*Rows*: 27,621), o número de variáveis (*Columns*: 62), o nome de cada coluna, o tipo de dado da variável e os registros iniciais contidos em cada variável. Veja o *script* e replique-o em seu `RStudio`:

```
# Visualizando o número de linhas e colunas com a função `glimpse()`  
glimpse(base)
```

```
#> Rows: 27,621
#> Columns: 62
#> $ NU_NOTIFIC <chr> "7671320", "0855803", "8454645", "3282723", "9799526", "727...
#> $ TP_NOT <int> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,...
#> $ ID_AGRAVO <chr> "A509", "W64", "X58", "A90", "B19", "A90", "A90", "Y09", "A...
#> $ CS_SUSPEIT <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ IN_AIDS <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ CS_MENING <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ DT_NOTIFIC <date> 2012-04-11, 2010-09-17, 2010-10-19, 2008-04-14, 2011-06-20...
#> $ SEM_NOT <chr> "201215", "201037", "201042", "200816", "201125", "200807", ...
#> $ NU_ANO <int> 2012, 2010, 2010, 2008, 2011, 2008, 2007, 2011, 2008, 2011,...
#> $ SG_UF_NOT <int> 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61,...
#> $ ID_MUNICIP <int> 610213, 610213, 610213, 610213, 610213, 610213, 610213, 610...
#> $ ID_REGIONA <lg> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ ID_UNIDADE <int> 256100, 180142, 559191, 180142, 480722, 570404, 319816, 289...
#> $ DT_SIN_PRI <date> 2012-04-05, 2010-09-09, 2010-10-19, 2008-04-11, 2011-04-02...
#> $ SEM_PRI <chr> "201214", "201036", "201042", "200815", "201113", "200806", ...
#> $ DT_NASC <date> 2012-04-04, 1988-04-23, 1971-03-25, 1928-05-29, 2002-09-18...
#> $ NU_IDADE_N <int> 2001, 4022, 4039, 4079, 4008, 4054, 4032, 4014, 4037, 4011,...
#> $ CS_SEXO <chr> "M", "M", "M", "F", "M", "F", "F", "F", "F", "F", "M", "M", ...
#> $ CS_GESTANT <int> 6, 6, 6, 9, 6, 9, 9, 9, 6, 5, 6, 6, 6, 6, 9, 9, 9, 9, 6, 5,...
#> $ CS_RACA <int> 4, 1, NA, 4, 4, 9, NA, 1, 9, 4, NA, 9, NA, NA, 9, NA, 1, 9,...
#> $ CS_ESCOL_N <chr> "10", NA, NA, "02", "01", "09", NA, "09", "09", "01", "10", ...
#> $ SG_UF <int> 33, 33, 33, 33, 33, 33, 33, 33, 33, 33, 33, 33, 33, 33, 33, 33,...
#> $ ID_MN_RESI <int> 610213, 610213, 610250, 610213, 610250, 610213, 610213, 610...
#> $ ID_RG_RESI <lg> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ ID_DISTRICT <chr> "05", "05", "05", "01", "01", "04", "05", "04", "01", "04", ...
#> $ ID_BAIRRO <chr> "020", "019", "020", "001", "001", "014", "020", "012", "00...
#> $ ID_LOGRADO <lg> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ ID_GEO1 <lg> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ ID_GEO2 <lg> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ CS_ZONA <int> 1, 1, NA, 1, 1, 1, 1, 1, 1, 1, 1, 1, NA, 1, 1, 1, 1, 1, ...
#> $ ID_PAIS <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
#> $ NDUPLIC_N <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ IN_VINCULA <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ DT_INVEST <date> NA, NA, 2010-10-19, 2008-04-14, 2011-06-20, NA, NA, NA, 20...
#> $ ID_OCUPA_N <chr> NA, NA, NA, NA, "999991", NA, NA, NA, NA, NA, NA, NA, NA, NA, N...
#> $ CLASSI_FIN <int> NA, NA, NA, 5, 1, 8, 8, 1, 1, NA, 8, 1, 8, 8, 1, NA, 1, 8, ...
#> $ CRITERIO <int> NA, NA, NA, 1, NA, NA, NA, NA, 2, NA, NA, 1, NA, NA, 2, NA,...
#> $ TPAUTOCTO <int> NA, NA, NA, NA, NA, NA, NA, NA, 1, NA, NA, 1, NA, NA, NA, N...
#> $ COUFINF <int> NA, NA, NA, NA, NA, NA, NA, NA, 61, NA, NA, 61, NA, NA, NA,...
#> $ COPAISINF <int> 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,...
#> $ COMUNINF <int> NA, NA, NA, NA, NA, NA, NA, NA, 610213, NA, NA, 610213, NA,...
#> $ CODISINF <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
#> $ CO_BAINFC <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 84, 0, 0, 0...
```


3.1 Selecionado as variáveis que serão analisadas

Você deve verificar que o objeto `{base}` foi criado a partir do arquivo `{NINDINET.dbf}`, que possui 62 variáveis. No entanto, para o exercício de análise do Estado de Rosas utilizaremos apenas as variáveis de interesse. Assim faremos um exercício neste tópico no qual reduziremos as colunas desta tabela para apenas 6 colunas com as variáveis epidemiológicas necessárias: Data da notificação (`DT_NOTIFIC`), Data de nascimento (`DT_NASC`), Sexo (`CS_SEX0`), Raça/cor (`CS_RACA`), Código do município de residência (`ID_MN_RESI`) e Código do CID10 do agravo notificado (`ID_AGRAVO`).

Para esta ação utilizaremos a função `select()`, que tem por objetivo selecionar variáveis de interesse para serem analisadas. Os argumentos principais da função `select()` são:

- a tabela de origem dos dados (*data.frame, tibble, etc*),
- os nomes das colunas que devem ser selecionadas.

Após aplicá-la, você terá como resultado uma tabela apenas com as variáveis escolhidas, ou seja, um recorte contendo todos os registros (linhas e/ou valores) do banco de dados, porém apenas com as seis variáveis (colunas) selecionadas.

Vamos praticar! Criaremos uma nova tabela apenas com seis variáveis escolhidas que chamaremos de `{base_menor}`. Esta tabela armazenará o objeto `{base}` que possui o banco de dados NINDINET (`{NINDINET.dbf}`) com as notificações do Estado de Rosas. Execute o seguinte comando e replique-o em seu RStudio:

```
# Criando a tabela {base_menor} e selecionado colunas específicas
base_menor <- select(base, DT_NOTIFIC, DT_NASC, CS_SEX0, CS_RACA, ID_
MN_RESI, ID_AGRAVO)
```

Agora, visualize em seu console as primeiras linhas de sua nova tabela, executando o comando. Replique-o em seu **RStudio**:

```
# Utilizando a função `head()` para visualizar colunas específicas da tabela {`base_menor`}
head(base_menor)
```

```
#>   DT_NOTIFIC   DT_NASC CS_SEXO CS_RACA ID_MN_RESI ID_AGRAVO
#> 1 2012-04-11 2012-04-04      M      4    610213    A509
#> 2 2010-09-17 1988-04-23      M      1    610213    W64
#> 3 2010-10-19 1971-03-25      M     NA    610250    X58
#> 4 2008-04-14 1928-05-29      F      4    610213    A90
#> 5 2011-06-20 2002-09-18      M      4    610250    B19
#> 6 2008-02-12 1953-08-01      F      9    610213    A90
```

Observe que a sua nova tabela possui apenas as colunas indicadas pela função `select()`.

3.2 Simplificando o fluxo de análises

Ao utilizar uma linguagem de programação como **R** para analisar dados, quase sempre precisaremos reaproveitar o resultado de uma análise anterior para realizar a análise seguinte, ou seja, utilizamos o *output* de um código anterior como o **argumento** principal de uma próxima análise. Esta ação é importante, pois tornará seu código menor e mais prático, evitando erros de análises quando precisamos repetir comandos já utilizados.

Observe no *script* abaixo como utilizaremos o *output* da função `select()`, que selecionamos algumas colunas, para visualizar as primeiras linhas usando a função `head()`. Estamos aninhando estas funções e, assim, teremos o mesmo resultado. Replique este comando em seu **RStudio**:

```
# Criando a tabela {`base_menor`} selecionado colunas específicas
base_menor <- select(base, DT_NOTIFIC, DT_NASC, CS_SEXO, CS_RACA, ID_
MN_RESI, ID_AGRAVO)

# Utilizando a função `head()` aninhada com a função `select()`
head(select(base, DT_NOTIFIC, DT_NASC, CS_SEXO, CS_RACA, ID_MN_RESI,
ID_AGRAVO))
```

Em análises como a que estamos trabalhando, à medida que precisamos realizar ações, o código fica maior e podemos nos perder entre tantas vírgulas, nomes de funções e parênteses. Como solução, no R podemos utilizar uma proposta de *sintaxe* (forma curta de escrever os comandos) implementada pela utilização de operadores chamados *pipes*, conforme descreve a tabela abaixo:

| Pipe | Descrição |
|------|---|
| > | nativo do R, carregado ao abrir uma nova sessão. |
| %>% | necessita do pacote <code>magrittr</code> , carregado no ambiente do <code>tidyverse</code> |

Nos exemplos deste curso iremos utilizar o operador `|>`, por dispensar o carregamento de pacotes! É importante destacar que o seu uso foi implementado a partir da versão 4.1 do R. Em códigos produzidos antes desta versão, será comum você encontrar o operador `%>%`, sendo necessário carregar o pacote `magrittr` ou o conjunto de pacotes pelo `tidyverse`.

Utilizando estes *pipes* (`|>` ou `%>%`) podemos encadear diferentes operações em que o objeto **à esquerda** do *pipe* (o *output* anterior) será utilizado como o primeiro argumento da função **à direita** do *pipe*, sem que seja necessário digitá-la novamente (isto é, o primeiro argumento é omitido).

Observe o *script* abaixo, em que criaremos o objeto `{base_menor}`, a partir do *data.frame* `{base}` utilizando o pipe (`|>`).Replique estas linhas em seu RStudio:

```
# Criando o objeto {`base_menor`} a partir do objeto {`base`} e  
# inserindo o pipe `|>` para encadear a seleção de variáveis  
base_menor <- base |>  
  select(DT_NOTIFIC, DT_NASC, CS_SEXO, CS_RACA, ID_MN_RESI, ID_AGRAVO)
```

Perceba que agora o objeto {`base`} ficou antes do operador `|>` e, em seguida, utilizamos a função `select()`, fornecendo diretamente os argumentos que viriam na sequência, isto é, o nome das colunas a serem selecionadas.

Agora acrescente a função `head()` para visualizar o resultado da operação. Replique o *script* abaixo em seu RStudio:

```
# Criando a {`base_menor`}  
base_menor <- base |>  
  
# Selecionando as variáveis que queremos utilizar com a função `select()`  
select(DT_NOTIFIC, DT_NASC, CS_SEXO, CS_RACA, ID_MN_RESI, ID_AGRAVO)  
|>  
  
# Acrescentando a função `head()` ao final do código após um pipe `|>`  
head()  
  
# Visualizando a o objeto {`base_menor`} criado com as primeiras linhas selecionadas  
# pela função `head()` e com colunas específicas, selecionadas pela função `select()`  
base_menor
```

```
#>   DT_NOTIFIC   DT_NASC CS_SEXO CS_RACA ID_MN_RESI ID_AGRAVO  
#> 1 2012-04-11 2012-04-04      M      4    610213    A509  
#> 2 2010-09-17 1988-04-23      M      1    610213     W64  
#> 3 2010-10-19 1971-03-25      M     NA    610250     X58  
#> 4 2008-04-14 1928-05-29      F      4    610213     A90  
#> 5 2011-06-20 2002-09-18      M      4    610250     B19  
#> 6 2008-02-12 1953-08-01      F      9    610213     A90
```

Como visto, os comandos acima selecionaram as colunas de interesse: Data da notificação (`DT_NOTIFIC`), Data de nascimento (`DT_NASC`), Sexo (`CS_SEXO`), Raça/cor (`CS_RACA`), Código do município de residência (`ID_MN_RESI`), Código do CID10 do agravo notificado (`ID_AGRAVO`) e, então, apresentam em seu console as linhas iniciais da tabela gerada.

Acompanhe as próximas seções deste módulo em que utilizaremos o pipe (`|>`) para analisar dados de vigilância em saúde.

4. Preparando as variáveis para análise de dados de vigilância

Com o **R** poderemos construir as tabelas personalizadas para nossas análises. Nesta seção você irá conhecer algumas maneiras de selecionar e transformar variáveis de interesse para vigilância em saúde.

4.1 Excluindo colunas na tabela

Quase sempre o profissional de vigilância necessita excluir colunas e reorganizar a sua ordem na tabela, e você já aprendeu neste curso que podemos selecionar colunas no **R** utilizando a função `select()`. Agora, utilizaremos esta função para excluir colunas.

Esta etapa é importante, pois permitirá excluir dados que não serão utilizados em nossa análise, tornando nossa tabela menor e mais “leve”, reduzindo o tempo gasto para seu processamento com **R**.

Para isso, a função `select()` deverá receber a inclusão do operador lógico subtração ("-"). A subtração deverá ser acrescentada antes do nome da coluna a ser excluída.

Para praticarmos excluirmos a variável de data de nascimento (`DT_NASC`) da tabela `{base_menor}` criada anteriormente. Acompanhe o passo a passo para excluirmos a variável `DT_NASC` do *data.frame* `{base_menor}` e replique-o em seu `RStudio`:

```
base_menor |>
```

```
# Excluindo a variável "DT_NASC" da tabela {`base_menor`}  
select(-DT_NASC) |>
```

```
# Utilizando a função `head()` para visualizar as primeiras linhas da tabela após a exclusão  
head()
```

```
#>   DT_NOTIFIC CS_SEXO CS_RACA ID_MN_RESI ID_AGRAVO  
#> 1 2012-04-11      M       4    610213    A509  
#> 2 2010-09-17      M       1    610213     W64  
#> 3 2010-10-19      M      NA    610250     X58  
#> 4 2008-04-14      F       4    610213     A90  
#> 5 2011-06-20      M       4    610250     B19  
#> 6 2008-02-12      F       9    610213     A90
```

Observe em seu output que o resultado da execução possui uma tabela com todas as colunas presentes em `{base_menor}`, exceto a variável `DT_NASC`. Fácil, não é mesmo?!



Atenção

Lembre-se de conferir todos os símbolos, palavras e pontos do seu código! O `R` retorna erros quando seu código está escrito de forma incorreta.

4.2 Inserindo novas colunas à tabela

Outra ação corriqueira quando analisamos dados de vigilância é adicionarmos colunas ao nosso banco de dados, seja com o resultado de algum cálculo, com informações de confirmação de um caso suspeito, ou ainda uma coluna com o status de uma investigação de surto. Para criar uma variável no R utilizamos a função `mutate()`. Seguiremos, portanto, o seguinte passo a passo:

1. definir o nome da coluna;
2. utilizar o sinal de igual (=) e;
3. escrever os comandos para incluir os valores que preencherão a nova coluna.

Lembre-se que estamos analisando a situação das notificações de casos no Estado de Rosas. Esta etapa apoiará nossa avaliação de sensibilidade do sistema de vigilância em saúde, pois criaremos uma nova variável que armazena valores com o tempo (em dias) que está levando para que uma notificação seja digitada no Sinan Net, ou seja, o **tempo em dias de atraso da digitação das notificações do Estado de Rosas**.

Para isso, criaremos a variável que daremos o nome de `TEMPO_DIGITA`, que será a diferença em dias entre a coluna com valores da data de digitação (`DT_DIGITA`) e a coluna data de notificação (`DT_NOTIFIC`). Como padrão, o R adicionará a nova variável criada ao final da tabela (última coluna).

Primeiro criamos um objeto que será chamado `{base_menor_2}`. Ele receberá as colunas selecionadas e seus respectivos registros da tabela `{base}` (criada anteriormente a partir do arquivo `{NINDINET.dbf}`). Observe o *script* abaixo e execute os comandos em seu `RStudio`:


```
# Criando a {'base_menor_2'}
base_menor_2 <- base |>

# Selecionando as variáveis que queremos utilizar com a função 'select()'
select(NU_NOTIFIC, ID_AGRAVO, DT_NOTIFIC, DT_DIGITA) |>

# Utilizando a função 'mutate()' para criar a nova coluna
mutate(TEMPO_DIGITA = DT_DIGITA - DT_NOTIFIC)
```

Agora, digite os códigos abaixo em seu *script* e pressione *Run* para visualizar como ficou sua nova variável criada:

```
# Utilizando a função 'head()' para visualizar as primeiras linhas do novo objeto
base_menor_2 |> head()
```

```
#>   NU_NOTIFIC ID_AGRAVO DT_NOTIFIC  DT_DIGITA TEMPO_DIGITA
#> 1    7671320      A509 2012-04-11 2012-11-09      212 days
#> 2    0855803       W64 2010-09-17 2010-11-17       61 days
#> 3    8454645       X58 2010-10-19 2011-03-14      146 days
#> 4    3282723       A90 2008-04-14 2008-04-24       10 days
#> 5    9799526       B19 2011-06-20 2011-09-14       86 days
#> 6    7275624       A90 2008-02-12 2008-02-26       14 days
```

Veja que esta nova tabela possui a coluna **TEMPO_DIGITA** no final, e suas linhas são a diferença da data - em dias - de digitação e de notificação. Após o número de dias, você verá a informação **days**, indicando que esta variável se trata de uma variável de tempo (classe = **difftime**).

Para fazer cálculos matemáticos e novas análises, precisaremos padronizar o tipo de dado contido na variável **TEMPO_DIGITA** e, aqui, optaremos pelo tipo numérico (*numeric*, em inglês). Neste caso, é possível alterar o tipo de dado da variável atribuindo-lhe o mesmo nome. Observe os *script* abaixo e replique-o em seu computador:

```
base_menor_2 |>
```

```
# Transformando a coluna "TEMPO_DIGITA" da tabela {`base_menor_2`}
# no tipo numérico com a função `as.numeric()` dentro da função `mutate()`
mutate(TEMPO_DIGITA = as.numeric(TEMPO_DIGITA)) |>

# Utilizando a função `head()` para visualizar as primeiras linhas
head()
```

```
#>   NU_NOTIFIC ID_AGRAVO DT_NOTIFIC  DT_DIGITA TEMPO_DIGITA
#> 1    7671320      A509 2012-04-11 2012-11-09          212
#> 2    0855803       W64 2010-09-17 2010-11-17           61
#> 3    8454645      X58 2010-10-19 2011-03-14          146
#> 4    3282723      A90 2008-04-14 2008-04-24           10
#> 5    9799526      B19 2011-06-20 2011-09-14           86
#> 6    7275624      A90 2008-02-12 2008-02-26           14
```

Visualize seu *output* e perceba que a coluna **TEMPO_DIGITA** é composta agora apenas por números (**numeric**), e não mais uma medida de tempo (**difftime**), nos apoiando a compreender o tempo que está levando para que as notificações de agravos do município de Rosas sejam digitadas no Sinan Net, em dias.

4.3 Limpando caracteres e padronizando colunas

Em análise de dados, algumas vezes não é possível reconhecer de imediato o formato de colunas ou mesmo se torna complicado utilizar **caracteres** como acentos, espaço entre as palavras, símbolos e outros, até mesmo nas funções de código.

Para resolver estes problemas, quase sempre necessitamos padronizar o nome das colunas do nosso banco de dado, limpando seus nomes e formatando-os para que haja uma fácil manipulação dos dados.

Para executarmos estas ações com o R, utilizamos o pacote `janitor` e sua função `clean_names()`. Esta função tornará todas as variáveis minúsculas e retirará os caracteres como acentos e os espaços entre as palavras, quando houver.

Para esta etapa, utilizaremos o banco de dados do tipo `.csv` que contêm os códigos do CID-10 exportados diretamente do site do Datasus: o `{CID-10-CATEGORIAS.csv}`. Este arquivo é muito utilizado para agrupar categorias de doenças, possibilitando a recategorização de CIDs em todos os bancos de dados do Datasus. Você o encontrará disponível no menu lateral “Arquivos”, no Ambiente Virtual do curso. Acompanhe o passo a passo abaixo e replique-o em seu RStudio:

1. Primeiro, importe o arquivo `{CID-10-CATEGORIAS.csv}`, armazenando-o no objeto do tipo `data.frame` `{cid10_categorias}`. Replique o *script* abaixo para importá-lo para o seu R:

```
# Importando o banco de dados { `CID-10-CATEGORIAS.CSV` } para o `R`  
cid10_categorias <- read_csv2('Dados/CID-10-CATEGORIAS.CSV')
```

2. Acrescente ao *script* a função `colnames()` para inspecionar o nome das colunas com o seguinte código:

```
# Inserindo a função `colnames()` para checar as variáveis  
colnames(cid10_categorias)
```

```
#> [1] "CAT"          "CLASSIF"      "DESCRICA0"   "DESCRABREV"  "REFER"  
#> [6] "EXCLUIDOS"
```

Observe o *output* em seu console. Você visualizará o vetor contendo os nomes de todas as colunas de sua tabela `cid10_categorias`.

3. Agora utilize a função `clean_names()` do pacote `janitor` ao rodar o seguinte código:

```
# Utilizando a função `clean_names()` para editar o nome das variáveis  
cid10_categorias_nova <- clean_names(cid10_categorias)  
  
# Visualizando as variáveis após a transformação  
colnames(cid10_categorias_nova)
```

```
#> [1] "cat"          "classif"      "descricao"   "descriabrev" "refer"  
#> [6] "excluidos"
```

Perceba que os nomes das variáveis estão totalmente em minúsculo, sem acentos e sem espaço entre as palavras.

Observe de forma detalhada a comparação dos nomes das colunas antes e após a transformação que realizamos com a função `clean_names()`.

Figura 2: Tabela comparativa da transformação das variáveis com a função `clean_names()`.

| Nome das colunas importadas | Nome após padronização da função <i>clean_names</i> |
|-----------------------------|---|
| CAT | cat |
| CLASSIF | classif |
| DESCRICA0 | descricao |
| DESCRABREV | describrev |
| REFER | refer |
| EXCLUIDOS | excluidos |



Atenção

Você deverá repetir este processo muitas vezes durante sua análise de dados. Esta etapa é fundamental para todos os próximos passos que daremos neste curso.

Ela nos permitirá ter segurança e não errar o nome das variáveis ao ter que digitá-los novamente para realizar filtros, seleções ou edições.

4.4 Filtrando colunas

Muito bem, já vimos como excluir colunas, transformá-las e renomeá-las para facilitar a análise dos dados. Agora, você irá aprender a filtrar os registros de uma tabela para obter as informações precisas para sua análise.

Para isso, utilizaremos a função `filter()` que permitirá filtrar os valores (linhas) da variável selecionada a partir de um ou mais critérios necessários. Para realizar esses filtros precisamos acrescentar os operadores lógicos indicando as ações de filtragem. Veja na lista abaixo quais são os principais operadores e seus usos:

| Operadores lógicos | Uso no filtro | Formas de uso |
|--|--|--|
| <code>==</code> | Filtra os registros iguais a um valor | <code>variável == 'valor'</code> ou <code>variável == 99</code> |
| <code>!=</code> | Filtra os registros diferentes de um valor | <code>variável != 'valor'</code> ou <code>variável != 99</code> |
| <code>></code> e <code>>=</code> | Filtra registros que uma variável seja maior / maior ou igual a um valor numérico | <code>variável > 99</code> ou <code>variável >= 99</code> |
| <code><</code> e <code><=</code> | Filtra registros que uma variável seja menor / menor ou igual a um valor numérico | <code>variável < 99</code> ou <code>variável <= 99</code> |
| <code>%in%</code> | Filtra os registros que os valores de uma variável sejam iguais a qualquer elemento de um conjunto de valores. | <code>variável %in% c('valor1', 'valor2', 'valor3')</code> |
| <code>!</code> | Usado sozinho, seleciona os registros que não atende ao critério definido | <code>!(variável == 'valor')</code> |

Vamos praticar! Utilizaremos o *data.frame* {base_menor_2}, criada anteriormente a partir da tabela {base}, para filtrar todos os registros que contenham o código da CID-10 referente à hepatite viral não especificada (B19) e que o tempo de digitação foi maior que sete dias. Execute o *script* abaixo em seu RStudio:

```
base_menor_2 |>

# Filtrando os CIDs igual a B19, da coluna "ID_AGRAVO"
# com tempo de digitação maior que sete dias
filter(ID_AGRAVO == "B19", TEMPO_DIGITA > 7) |>

# Utilizando a função `head()` para visualizar as primeiras linhas da tabela
head()
```

```
#>      NU_NOTIFIC ID_AGRAVO DT_NOTIFIC  DT_DIGITA TEMPO_DIGITA
#> 5      9799526      B19 2011-06-20 2011-09-14      86 days
#> 34     4218628      B19 2009-07-10 2010-05-11     305 days
#> 51     9943142      B19 2010-08-03 2010-09-24      52 days
#> 76     1856028      B19 2009-06-22 2009-07-09      17 days
#> 85      0025650      B19 2011-01-31 2011-07-25     175 days
#> 91      0733923      B19 2007-01-25 2007-03-28      62 days
```

Veja que, por padrão, quando se utiliza a vírgula para separar os diversos filtros que vamos utilizar `filter(ID_AGRAVO == "B19", TEMPO_DIGITA > 7)`, a função `filter()` realiza a ação considerando o primeiro critério **E** o segundo e assim sucessivamente.

Caso exista necessidade de que o filtro seja um critério do tipo **OU**, você necessitará utilizar o operador barra horizontal (`|`). Observe no exemplo abaixo que o filtro está selecionando registros de notificação de hepatite (CID B19) **OU** de leptospirose (CID A279) **OU** malária (CID B54):

```
base_menor_2 |>
```

```
# Filtrando os registros iguais a B19 OU A279 OU B54 na coluna "ID_AGRAVO"  
filter(ID_AGRAVO == "B19" | ID_AGRAVO == "A279" | ID_AGRAVO == "B54") |>
```

```
# Utilizando a função `head()` para visualizar as primeiras 20 linhas da tabela  
head(20)
```

```
#>      NU_NOTIFIC ID_AGRAVO DT_NOTIFIC  DT_DIGITA TEMPO_DIGITA  
#> 5      9799526      B19 2011-06-20 2011-09-14      86 days  
#> 34     4218628      B19 2009-07-10 2010-05-11     305 days  
#> 51     9943142      B19 2010-08-03 2010-09-24      52 days  
#> 76     1856028      B19 2009-06-22 2009-07-09      17 days  
#> 85      0025650      B19 2011-01-31 2011-07-25     175 days  
#> 91      0733923      B19 2007-01-25 2007-03-28      62 days  
#> 106     8181226      B19 2009-07-06 2009-07-23      17 days  
#> 112     5619327     A279 2009-12-30 2010-01-21      22 days  
#> 128     4231128      B19 2009-08-04 2010-05-04     273 days  
#> 157     6644624      B19 2007-01-17 2007-04-12      85 days  
#> 160     4232928      B19 2009-09-29 2009-11-23      55 days  
#> 169     4360123      B19 2012-12-27 2013-01-28      32 days  
#> 172     1821124      B54 2008-08-11 2008-08-29      18 days  
#> 175     1449327      B19 2010-05-13 2010-10-25     165 days  
#> 190     1448227      B19 2009-08-05 2010-09-21     412 days  
#> 198     1250223      B19 2007-02-15 2007-06-05     110 days  
#> 212     8469526      B19 2008-01-18 2008-07-03     167 days  
#> 220     9777227      B19 2009-04-22 2009-05-21      29 days  
#> 231     9941142      B19 2010-08-03 2010-09-21      49 days  
#> 242     2338425      B19 2008-01-14 2008-08-11     210 days
```

Também é possível utilizar o operador (`%in%`) nesta situação, quando a mesma variável (`ID_AGRAVO`) pode conter vários valores passíveis de seleção (filtros). Nesta situação, portanto, selecionamos os agravos dentro do conjunto dado pela função `c()`.

Acompanhe o *script* abaixo e replique-o:

```
base_menor_2 |>
```

```
# Filtrando os agravos utilizando o operador `%in%`  
filter(ID_AGRAVO %in% c("B19", "A279", "B54")) |>
```

```
# Utilizando a função `head()` para visualizar as primeiras 20 linhas da tabela  
head(20)
```

```
#>      NU_NOTIFIC ID_AGRAVO DT_NOTIFIC  DT_DIGITA TEMPO_DIGITA  
#> 5      9799526      B19 2011-06-20 2011-09-14      86 days  
#> 34     4218628      B19 2009-07-10 2010-05-11     305 days  
#> 51     9943142      B19 2010-08-03 2010-09-24      52 days  
#> 76     1856028      B19 2009-06-22 2009-07-09      17 days  
#> 85      0025650      B19 2011-01-31 2011-07-25     175 days  
#> 91      0733923      B19 2007-01-25 2007-03-28      62 days  
#> 106     8181226      B19 2009-07-06 2009-07-23      17 days  
#> 112     5619327     A279 2009-12-30 2010-01-21      22 days  
#> 128     4231128      B19 2009-08-04 2010-05-04     273 days  
#> 157     6644624      B19 2007-01-17 2007-04-12      85 days  
#> 160     4232928      B19 2009-09-29 2009-11-23      55 days  
#> 169     4360123      B19 2012-12-27 2013-01-28      32 days  
#> 172     1821124      B54 2008-08-11 2008-08-29      18 days  
#> 175     1449327      B19 2010-05-13 2010-10-25     165 days  
#> 190     1448227      B19 2009-08-05 2010-09-21     412 days  
#> 198     1250223      B19 2007-02-15 2007-06-05     110 days  
#> 212     8469526      B19 2008-01-18 2008-07-03     167 days  
#> 220     9777227      B19 2009-04-22 2009-05-21      29 days  
#> 231     9941142      B19 2010-08-03 2010-09-21      49 days  
#> 242     2338425      B19 2008-01-14 2008-08-11     210 days
```

Pronto, você armazenou na tabela ou *data.frame* {base_menor_2} apenas os registros que contenham notificações de hepatites virais (CID B19), leptospirose (CID A279) e malária (CID B54).



A qualquer momento em que desejar obter uma tabela com dados que contenham todas as notificações do Estado de Rosas você deverá utilizar o objeto `{base}` criado no início deste módulo.

Lembre-se que o objeto `{base}` é do tipo *data.frame* e armazenou os dados importados do banco de dados `{NINDINET.dbf}`, disponibilizado no Ambiente Virtual do curso.

4.5 Unindo colunas

Ainda na etapa de limpeza e organização de dados, frequentemente o profissional de vigilância precisa agrupar registros para suas análises. A variável idade é uma das variáveis epidemiológicas mais utilizadas, por exemplo, para divisão de grupos por faixa etária em quase todas as análises demográficas.

Na linguagem R, a função `group_by()` é o verbo que utilizaremos para agrupamento dos registros de acordo com a coluna especificada, sendo a variável o principal argumento da função. Aqui, destacamos que é muito frequente seu uso com outras funções junto ao banco de dados.

Perceba que na expressão abaixo realizamos comandos de agrupamento (`group_by`) de todos os casos confirmados ou suspeitos notificados pela mesma doença ou agravamento (`ID_AGRAVO`). Em seguida realizamos um comando para contar (usando a função `count()`) quantos casos foram registrados com a mesma doença ou agravamento. Permaneceremos utilizando a tabela `{base_menor_2}` criada anteriormente. Rode o código a seguir em seu computador e verifique o resultado:

```
base_menor_2 |>
```

```
# Agrupando as notificações pelos agravos  
group_by(ID_AGRAVO) |>
```

```
# Contando a frequência de notificações por agravos  
count(ID_AGRAVO)
```

```
#> # A tibble: 63 × 2  
#> # Groups:   ID_AGRAVO [63]  
#>   ID_AGRAVO      n  
#>   <chr>      <int>  
#> 1 A010         2  
#> 2 A051         2  
#> 3 A059         2  
#> 4 A09          4  
#> 5 A169       2347  
#> 6 A279         49  
#> 7 A309        266  
#> 8 A35          4  
#> 9 A379         26  
#> 10 A509       329  
#> # ... with 53 more rows
```

O código digitado foi executado e retornou um objeto do tipo *tibble* e mostra três colunas: a primeira é uma coluna índice que marca o número da linha, não sendo uma variável do banco em si; a segunda é a **ID_AGRAVO** com todos os CID10 notificados (variável do tipo *character*, `<chr>`); e a **n** tem o número total de vezes que este agravo foi notificado (variável do tipo numérica, `<int>`).

Perceba também, que ao final do código, há as mensagens:

```
#> # ... with 53 more rows  
#> # i Use print(n = ...) to see more rows
```



Esses avisos aparecem porque o R retorna por padrão apenas 10 linhas da tabela para visualização. Caso queira que ele retorne mais linhas você deverá inserir o valor de linhas valor que deseja visualizar na função `print()` dentro de seu argumento `n =`. Observe o script abaixo como fazemos e reproduza em seu RStudio:

```
base_menor_2 |>  
  
# Agrupando as notificações pelos agravos  
group_by(ID_AGRAVO) |>  
  
# Contando a frequência de notificações por agravos  
count(ID_AGRAVO) |>  
  
# Visualizando as 20 primeiras linhas da tabela {`base_menor_2`}   
# usando a função `print()`  
print (n = 20)
```

```
#> # A tibble: 63 × 2
#> # Groups:   ID_AGRAVO [63]
#>   ID_AGRAVO      n
#>   <chr>      <int>
#> 1 A010         2
#> 2 A051         2
#> 3 A059         2
#> 4 A09          4
#> 5 A169       2347
#> 6 A279        49
#> 7 A309       266
#> 8 A35          4
#> 9 A379        26
#> 10 A509       329
#> 11 A510         1
#> 12 A513         3
#> 13 A53        121
#> 14 A530         1
#> 15 A539       234
#> 16 A60         32
#> 17 A63          7
#> 18 A630        76
#> 19 A64          5
#> 20 A779        12
#> # ... with 43 more rows
```

Agora já conseguimos saber que a leptospirose (CID10 = A279) aparece notificada por 49 vezes nesta tabela. Esse pequeno comando é mais simples, seguro e rápido que contar no Excel ou gerar uma tabela dinâmica.

Assim, todas as vezes em que for necessário realizar a contagem das frequências de uma variável epidemiológica, o profissional de vigilância poderá utilizar o verbo `group_by()` para fazê-lo em segundos.

4.6 Resumindo os valores de uma coluna

Outro aspecto importante do trabalho da vigilância é a capacidade de extrair dados e resumi-los. Assim, nesta etapa você irá aprender a função `summarise()`, similar à função `mutate()` utilizada anteriormente. A principal diferença entre as duas funções é a possibilidade de criação de novas colunas realizando operações de síntese e resumo, conjuntamente. Assim, a função `summarise()` possibilita o resumo de muitos valores em uma única linha.

Para praticar o uso do `summarise()` calcularemos **a média do tempo de atraso (em dias) em que as equipes de vigilância do Estado de Rosas têm digitado suas notificações** seguindo os seguintes passos:

1. utilizaremos como banco de dados a tabela `{base_menor_2}` criada a partir da tabela `{`base`}`,
2. em seguida, utilizaremos a função `group_by()` para contabilizar o número de casos por agravos ou doenças notificados, utilizando a função de contagem `n()` da variável `TEMPO_DIGITA`, criada anteriormente, e
3. por fim, utilizaremos a função `mean()`, a qual já vimos que calcula a média de uma coluna.

Agora observe o *script* abaixo e replique-o em seu RStudio:

```
base_menor_2 |>

# Agrupando as notificações pelos agravos
group_by(ID_AGRAVO) |>

# Utilizando a função `summarise()` para criar novas colunas de síntese
summarise(

  # Criando uma coluna de total, utilizando a função `n()`
  total_agravos = n(),

  # Criando uma coluna de média, utilizando a função `mean()`
  media_digita = mean(TEMPO_DIGITA)
)
```

```
#> # A tibble: 63 × 3
#>   ID_AGRAVO total_agravos media_digita
#>   <chr>          <int> <drtn>
#> 1 A010                2 122.00000 days
#> 2 A051                2  10.50000 days
#> 3 A059                2  27.00000 days
#> 4 A09                 4 193.50000 days
#> 5 A169             2347 151.67490 days
#> 6 A279                49  78.55102 days
#> 7 A309             266 165.64286 days
#> 8 A35                 4  70.75000 days
#> 9 A379                26  36.03846 days
#> 10 A509             329 130.58055 days
#> # ... with 53 more rows
```

O comando executado retornou quatro colunas:

- a primeira a coluna é a índice, que mostra o número da linha;
- a segunda é a `ID_AGRAVO` com todos os CID10 notificados (variável do tipo texto, `<chr>`);
- a coluna **total_agravos** corresponde ao número total de vezes que este agravo foi notificado (variável do tipo inteiro, `<int>`) e;
- por fim a variável que criamos **media_digita** (variável do tipo data, `<drtn>`).

Assim, com esta nova tabela conseguimos concluir que leptospirose (CID10 = A279) aparece notificada por 49 vezes e o tempo médio de digitação no sistema foi 78,55102 dias.

4.7 Modificando a ordem das colunas

No dia a dia é comum precisarmos modificar a ordem de apresentação dos valores de uma coluna como por exemplo ordenar idades do mais velhos para os mais novos, ou mesmo ordenar uma coluna por ordem alfabética, como de A-Z ou de Z-A. Para esta reordenação de valores em uma variável no R utilizamos a função `arrange()`.

Para executar esta função, precisamos apenas das colunas que queremos ordenar como argumento. Por padrão, essa função ordena do menor valor para o maior (ordem crescente). Caso exista a necessidade de ordenar uma coluna de forma decrescente, devemos também utilizar a função `desc()`, como no exemplo abaixo: `arrange(desc(media_digita))`.

Vejamos o mesmo exemplo anterior, mas agora adicionando o comando de ordenação decrescente para variável **media_digita**, apontando, assim, o agravo ou doença que leva o maior tempo para ser digitado no Estado de Rosas. Utilize os seguintes comandos:


```
base_menor_2 |>
```

```
# Agrupando as notificações pelos agravos  
group_by(ID_AGRAVO) |>
```

```
# Utilizando a função `summarise()` para criar novas colunas de síntese  
summarise(total_agravos = n(),  
           media_digita = mean(TEMPO_DIGITA)) |>
```

```
# Ordenando a tabela pela ordem decrescente da média de tempo de digitação  
arrange(desc(media_digita))
```

```
#> # A tibble: 63 × 3  
#>   ID_AGRAVO total_agravos media_digita  
#>   <chr>          <int> <drtn>  
#> 1 F99              3 666.6667 days  
#> 2 C80              1 560.0000 days  
#> 3 Z206            23 374.9130 days  
#> 4 H833            11 331.6364 days  
#> 5 N485             5 304.6000 days  
#> 6 Z209           500 293.6880 days  
#> 7 B09            109 283.8807 days  
#> 8 B42            150 269.0333 days  
#> 9 A630            76 263.8289 days  
#> 10 R36            77 261.9740 days  
#> # ... with 53 more rows
```

Como resultado você verá quatro colunas. A primeira com o índice com o número da linha, a segunda com a identificação do código do agravo, a terceira com o total de casos e a última com a média do tempo de digitação para cada agravo.

Veja que como as colunas estão ordenadas em ordem decrescente, com o retorno do comando é possível visualizar o maior valor primeiro (666,6667 dias), que se refere ao agravo relacionado ao trabalho, e os valores com a menor média de digitação aparecem ao final da coluna.

5. Avaliando a completude dos dados

Nesta etapa, vamos conhecer o preenchimento dos campos no registro da Ficha de Notificação Individual (FIN) do Sinan Net. Ao final deste tópico você será capaz de analisar a presença de variáveis em branco ou nulas, utilizando as seguintes funções do R: `is.na()`, `sum()` e `n()`.

Enquanto profissional de vigilância em saúde você deve estar acostumado a manipular formulários que têm campos de preenchimento em branco ou ignorados. Na epidemiologia o indicador de completude de uma variável é calculado a partir da proporção de campos preenchidos em relação ao total dos registros, em percentual.

Para visualizar de forma simples estes registros em uma coluna, vamos continuar utilizando a tabela criada a partir do banco de dados `{NINDINET.dbf}`, disponível no Ambiente Virtual do curso. Este banco foi armazenado no objeto `{base}` e agora vamos avaliar a completude da variável epidemiológica Raça/cor (`CS_RACA`).

Para isso utilizaremos a função `summarise()`, pois estamos resumindo algumas características da variável. Não se preocupe com o *script*, explicaremos mais adiante o passo a passo realizado.

Acompanhe o código abaixo e replique-o em seu RStudio:

```
base |>

# Utilizando a função `summarise()` para criar novas colunas de síntese
summarise(

  # Criando uma coluna com a soma de todos os registros da variável raça/cor devidamente preenchidos
  total_completo = sum(!is.na(CS_RACA)),

  # Criando uma coluna com o total de registros na tabela
  total_registros = n(),

  # Criando uma coluna com a soma de registros com a variável raça/cor em branco
  total_missing_raca = sum(is.na(CS_RACA)),

  # Criando uma coluna de porcentagem de completude (preenchimento)
  taxa_completude = (total_completo / total_registros) * 100
)
```

```
#>   total_completo total_registros total_missing_raca taxa_completude
#> 1           19326           27621             8295           69.9685
```

Observe em seu painel Console ou *output*, que foram retornadas quatro colunas:

- Em `total_completo` é indicado o número total de registros para a variável `CS_RACA` que não estão em branco (isto é, são diferentes de `NA`).
- Em `total_registros` é indicado o número total de registros, independentemente se estão preenchidos ou não.
- Em `total_missing_raca` é indicado o número de registros em branco apenas para a variável `CS_RACA`.
- Por fim, `taxa_completude` mostra a porcentagem de casos completos em relação ao total de casos registrados.

Vamos entender agora como conseguimos estes resultados, detalhando as funções:

A função `is.na()` indica em termos de verdadeiro/falso (TRUE/FALSE) quais registros estão em branco (isto é, são iguais a NA). Já a operação `!is.na()` calcula exatamente o oposto: quais registros estão preenchidos, isto é, diferentes de NA. Isto é indicado pelo operador (!) (ponto de exclamação), que inverte o resultado dos valores de verdadeiro e falso (TRUE e FALSE). Ao combinarmos estas funções com a função de soma (`sum()`), podemos calcular a soma do número de registros que se enquadram nos critérios avaliados.

Já a taxa de completude em porcentagem é calculada pela razão entre o total de registros preenchidos pelo total de registros, multiplicado por 100.



É possível realizar a análise de completude de variáveis utilizando outros pacotes no R. Um pacote muito utilizado é o `skmr`. Nele, há funções muito interessantes para verificação do preenchimento de variáveis, como:

- `n_complete`, que verifica o total de células preenchidas de uma variável;
- `n_missing`, que verifica o número de células em branco; e
- `complete_rate`, que faz uma razão entre o total células preenchidas pelo total de registros.

No exemplo abaixo vamos verificar o preenchimento da variável Raça/cor (`CS_RACA`), utilizada anteriormente, combinando a função `summarise()` com as funções do pacote `skmr`. Para isso, rode o seguinte código:

```
base |>

# Utilizando a função `summarise()` para criar novas colunas de síntese
summarise(

  # Criando uma coluna de total de registros com a variável raça/cor devidamente preenchida
  total_completo = n_complete(CS_RACA),

  # Criando uma coluna de total de registros da variável raça/cor em branco
  total_na = n_missing(CS_RACA),

  # Criando uma coluna de taxa de completude
  taxa_completude = complete_rate(CS_RACA) * 100
)
```

```
#>   total_completo total_na taxa_completude
#> 1           19326     8295           69.9685
```

Veja que foi possível conhecer os valores nulos ou em brancos com poucas linhas de comando. Agora você já pode realizar seus relatórios de completude dos bancos de dados sem dificuldades utilizando o software R.

5.1 Renomeando valores

Algumas variáveis nos sistemas de informações possuem uma categoria chamada “Ignorado”. Essa categoria se refere a dados que, por alguma eventualidade, não foram coletados no ato do preenchimento da notificação e por serem de preenchimento obrigatório são completados desta forma. Comumente, nos sistemas de informações, essa categoria é definida com o código 9 ou 99. Vejamos na Figura 3 um recorte da Ficha de Notificação Individual (FIN).

Figura 3: Início da Ficha de Notificação Individual (FIN).

| | | | | | |
|-------------------------------|--|--|--|---|--|
| Notificação Individual | 8 Nome do Paciente | | 9 Data de Nascimento | | |
| | 10 (ou) Idade 1 - Hora 2 - Dia 3 - Mês 4 - Ano | | 11 Sexo M - Masculino F - Feminino I - Ignorado | 12 Gestante 1-1º Trimestre 2-2º Trimestre 3-3º Trimestre 4- Idade gestacional Ignorada 5-Não 6- Não se aplica 9- Ignorado | |
| | 13 Raça/Cor 1-Branca 2-Preta 3-Amarela 4-Parda 5-Indígena 9- Ignorado | | 14 Escolaridade 0-Analfabeto 1-1ª a 4ª série incompleta do EF (antigo primário ou 1º grau) 2-4ª série completa do EF (antigo primário ou 1º grau) 3-5ª à 8ª série incompleta do EF (antigo ginásio ou 1º grau) 4-Ensino fundamental completo (antigo ginásio ou 1º grau) 5-Ensino médio incompleto (antigo colegial ou 2º grau) 6-Ensino médio completo (antigo colegial ou 2º grau) 7-Educação superior incompleta 8-Educação superior completa 9- Ignorado 10- Não se aplica | | |
| | 15 Número do Cartão SUS | | 16 Nome da mãe | | |

Notificação Individual

Observe que em todos os campos de preenchimento em múltipla escolha é possível selecionar “Ignorado”. Uma ação para garantir a consistência da análise de seus dados é realizar o agrupamento dos valores que estão nulos ou em branco (*missing*, em inglês) com os itens ignorados. Para esta etapa, realizamos a codificação de todos os valores com o código 9.

Para esta etapa, utilizaremos a função `replace_na()` do pacote `tidyr` para realizar a substituição dos valores em branco ou nulos por 9, combinada com a função `mutate()`.

Nesta tarefa, vamos permanecer manipulando a variável Raça/cor (`CS_RACA`) do objeto `{base}`, criada a partir do banco de dados `{NINDINET.dbf}`. Vejamos primeiro a contagem das categorias utilizando o seguinte comando em seu console:

```
base |>
```

```
# Contabilizando o número de registros conforme as categorias de preenchimento  
# da variável "CS_RACA"  
count(CS_RACA)
```

```
#>   CS_RACA    n  
#> 1      1 5298  
#> 2      2 1792  
#> 3      3   88  
#> 4      4 3799  
#> 5      5   40  
#> 6      9 8309  
#> 7     NA 8295
```

Perceba que há 8.309 registros com a variável Raça/cor preenchidos como **Ignorado** (código 9) e 8.295 com **<NA>** nulo ou em branco. Agora vamos agrupar o Ignorado (indicado pelo valor 9) com **NA** utilizando a função `replace_na()`. O primeiro argumento desta função é a coluna ao qual queremos aplicar a transformação, e o segundo argumento (`replace`) indica qual valor queremos que substitua o valor **NA**. Em nosso exemplo, queremos que seja substituído pelo valor 9.

Desta forma, execute os comandos do *script* abaixo em seu **RStudio**:

```
base |>
```

```
# Transformando os valores em branco em 9 na coluna "CS_RACA"  
mutate(CS_RACA = replace_na(CS_RACA, replace = 9)) |>  
  
# Contabilizando o número de registros conforme as categorias de preenchimento  
# da variável "CS_RACA"  
count(CS_RACA)
```

```
#>   CS_RACA      n
#> 1      1  5298
#> 2      2  1792
#> 3      3    88
#> 4      4  3799
#> 5      5    40
#> 6      9 16604
```

Encontramos 16.604 registros contabilizados como 9, ou seja, **Ignorado** e **NA**. Esse valor é igual aos 8.309 registros somados aos 8.295. Tente fazer a mesma operação com outras variáveis do banco de dados e renomeie com valores nulos!



É também possível que você necessite transformar os valores codificados como **Ignorado** em **NA**. Para isso, você deverá utilizar a função `na_if()` do pacote `dplyr`. O primeiro argumento desta função também deve indicar a coluna que queremos aplicar a transformação, enquanto o segundo indica qual valor queremos que seja convertido em **NA**.

Execute o código e veja abaixo como ficaria no exemplo anterior com a variável **Raça/cor**:

```
base |>

# Transformando os valores codificados como 9 em NA, na coluna "CS_RACA"
mutate(CS_RACA = na_if(CS_RACA, "9")) |>

# Contabilizando o número de registros conforme as categorias de preenchimento
# da variável "CS_RACA"
count(CS_RACA)
```



```
#>   CS_RACA      n
#> 1       1  5298
#> 2       2  1792
#> 3       3    88
#> 4       4  3799
#> 5       5    40
#> 6      NA 16604
```

Agora você tem 16.604 registros classificados como `<NA>`. Foi fácil, não é mesmo?!

6. Arrumando os dados de vigilância em saúde

Uma etapa importante para a produção de informações de qualidade é o cuidado com a organização e o armazenamento do dado coletado diariamente. Arrumar um dado significa garantir que houve uma forma padronizada de conectar a estrutura (formato) de um conjunto de dados à sua semântica (significado).

Dados bem estruturados servem para:

- fornecer dados seguros para o processamento e análise de dados por softwares;
- revelar informações e facilitar a percepção de padrões.

Neste tópico vamos conhecer técnicas para arrumar nossos dados e analisá-los de forma eficiente e segura.

6.1 Organizando dados em linhas e colunas (dados retangulares)

Os objetos mais comuns que armazenam dados possuem uma estrutura de duas dimensões: linhas e colunas. Por exemplo, na *matriz* que calcula taxa de ataque em casos de surtos temos linhas e colunas que armazenam um mesmo tipo de dado, ou seja, apenas números, mas também poderá armazenar outros formatos possíveis, como textos. Já em um *data.frame*, uma coluna pode conter dados numéricos, outra dados textuais e outra datas.

Você perceberá que há uma organização que nos remete a uma forma geométrica bem definida e estruturada, como a de um retângulo. Por definição, dados organizados em linhas e colunas são chamados dados retangulares.

Já os dados não retangulares seriam aqueles armazenados em uma estrutura diferente como, por exemplo, texto de um discurso, ou até mesmo o campo de observação das fichas de notificação compulsórias.

Assim, dados retangulares são os dados no “formato arrumado” que atendem às seguintes regras:

1. cada variável está em uma coluna;
2. cada observação* corresponde a uma linha;
3. cada valor corresponde a uma célula;
4. cada tipo de unidade observacional deve compor uma tabela.

**Como sinônimo de observações você pode encontrar os termos: registros, casos, exemplos, instâncias ou amostras, dependendo da área de aplicação.*

Na Vigilância em Saúde, o uso de dados no formato retangular é comum e quase sempre necessário, sendo a forma de armazenamento de todos os Sistemas de Informações em Saúde e, inclusive, de outras fontes de registro comuns na área da saúde, como prontuários, formulários do *REDcap*, *formSUS* ou mesmo *google forms*.

Vamos lá, enquanto profissional de vigilância você necessita buscar informações sobre o acompanhamento de pacientes na atenção primária em saúde. Para isso, irá consultar o registro dos atendimentos realizados no e-SUS AB. Na consulta ao sistema você encontrou o registro de quatro pessoas e seus atendimentos. A partir disso, você abriu uma planilha de Excel e criou uma tabela na qual inseriu o número de identificação da ficha de atendimento, a idade e o sexo biológico de cada um dos pacientes.

Na Figura 4 exibimos a representação visual da planilha criada por você e que aqui será utilizada como nosso banco de dados.

Figura 4: Banco de dados com o número de identificação da ficha de atendimento, a idade e o sexo biológico de cada um dos pacientes.

| Número da ficha | Idade | Sexo |
|-----------------|-------|------|
| 001 | 24 | M |
| 002 | 32 | F |
| 003 | 38 | F |
| 004 | 12 | M |

É possível perceber o formato retangular dos dados em que cada coluna se refere a diferentes tipos de dados (a primeira variável, por exemplo, é composta por um número ordenado, a segunda por um número inteiro e a terceira por texto).

Agora, observe a Figura 5: a Figura 5A tem como destaque a coluna com a variável Idade, representando a idade de um paciente em uma única coluna. Na Figura 5B, a linha em destaque se refere a um único paciente, identificado pelo número da ficha de registro, e a Figura 5C evidencia a célula em destaque com um único valor de idade de um único paciente.

Figura 5: Tabelas identificando coluna (A), linha (B) e valor (C).

A

| Número da ficha | Idade | Sexo |
|-----------------|-------|------|
| 001 | 24 | M |
| 002 | 32 | F |
| 003 | 38 | F |
| 004 | 12 | M |

B

| Número da ficha | Idade | Sexo |
|-----------------|-------|------|
| 001 | 24 | M |
| 002 | 32 | F |
| 003 | 38 | F |
| 004 | 12 | M |

C

| Número da ficha | Idade | Sexo |
|-----------------|-------|------|
| 001 | 24 | M |
| 002 | 32 | F |
| 003 | 38 | F |
| 004 | 12 | M |

Em todos os casos apresentados perceba que os dados são organizados da forma mais correteira, em bancos de dados utilizados no seu dia a dia, quase sempre no formato retangular.

6.2 Modificando o formato de um banco de dados

Os dados retangulares podem apresentar formatos diferentes, sendo os mais comuns largo e longo (em inglês *wide* e *long*, respectivamente). O formato longo se refere a um visual no qual a base aparenta ter muitas linhas em relação às colunas, e o formato largo se refere às tabelas que aparentam ter mais colunas, estando assim muito “larga” em relação às suas linhas.

Na linguagem R, vários pacotes e funções são executadas de forma mais eficiente quando os dados são organizados no formato longo (muitas linhas em relação às colunas). Alguns deles, como o pacote utilizado para a visualização de gráficos **ggplot2**, são essenciais para uma análise exploratória de dados. Além disso, a própria prática de explorar os dados torna-se mais fluida quando há uma padronização uniforme nas bases de dados.

Como profissional de vigilância em saúde, você está acostumado com bases de dados que muitas vezes não estão organizadas no formato longo ou largo ou, ainda, que sigam as regras citadas acima. Vejamos o exemplo abaixo.

Você se lembra do banco de dados disponibilizado pela equipe do Setor de Imunização do Estado de Rosas com os dados de cobertura vacinal contra Hepatite B em crianças de até 30 dias de idade? Vamos utilizá-lo novamente aqui! Estes dados correspondem à vacinação realizada de 2016 a 2020 nos municípios de Prímula e Antúrio, da região norte de Rosas, exportados pela equipe junto ao módulo de Imunização do Tabnet e alterados a partir de um arquivo com extensão do tipo `.csv` {cobertura_hepatiteb_rosas_2016_2020_A.csv}. Este arquivo está disponível no menu lateral “Arquivos” deste módulo.

Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Importando o banco de dados { `cobertura_hepatiteb_rosas_2016_2020_A.csv` } para o `R`  
dados <- read_csv(file = "Dados/cobertura_hepatiteb_rosas_2016_2020_A.csv")
```

```
#> # A tibble: 2 × 6  
#>   Municipio `Ano 2016` `Ano 2017` `Ano 2018` `Ano 2019` `Ano 2020`  
#>   <chr>      <chr>      <chr>      <chr>      <chr>      <chr>  
#> 1 Primula   99.55%    90.09%    91.54%    86.75%    69.32%  
#> 2 Anturio  79.97%   113.83%   131.06%   115.53%   95.16%
```

Perceba que a equipe do Setor de Imunização mantém esta base no formato largo. A tabela produzida a partir do *data.frame* possui seis colunas e duas linhas, onde cada linha se refere a um município específico. Entretanto, há cinco colunas que se referem à mesma variável (Ano), colunas que representam a mesma característica: cobertura vacinal.

Para analisar as coberturas vacinais por ano, necessitaremos que todos os dados da tabela {dados} estejam em uma única coluna. Isso porque precisamos facilitar a organização dos dados em um formato que se possa analisá-los temporalmente, considerando por tanto que cada coluna faz referência a um ano. Fique tranquilo, explicaremos de forma detalhada mais adiante estas etapas.

Dessa forma, modificaremos o formato **largo** (*wide*) para uma tabela em formato **longo** (*long*) com a função de **pivotagem** no R.

A **pivotagem** é uma técnica que se refere à mudança de formato de uma base, sendo uma das principais operações de transformação de dados. Basicamente a **pivotagem** transforma nosso banco de dados do formato largo para longo (ou vice-versa) e, dessa forma, alterando suas dimensões.

Observe a Figura 6 em que é ilustrada a pivotagem de uma tabela do formato **largo** (*wide*) para **longo** (*long*).

Figura 6: Transformação de tabelas do formato longo para largo.

Código

```
dados |>
  pivot_longer(
    cols = c("Ano 2016", "Ano 2017", "Ano 2018", "Ano 2019", "Ano 2020"),
    names_to = "Ano",
    values_to = "Cobertura Vacinal contra Hepatite B"
  )
```

Do formato Wide

| Município | Ano 2016 | Ano 2017 | Ano 2018 | Ano 2019 | Ano 2020 |
|-----------|----------|----------|----------|----------|----------|
| Primula | 99,55% | 90,09% | 91,54% | 86,75% | 69,32% |
| Anturio | 79,97% | 113,83% | 131,06% | 115,53% | 95,16% |

→ **names_to** → Ano

→ **values_to** → Cobertura Vacinal contra Hepatite B

Para formato Long

| Município | Ano | Cobertura Vacinal contra Hepatite B |
|-----------|----------|-------------------------------------|
| Primula | Ano 2016 | 99,55% |
| Primula | Ano 2017 | 90,09% |
| Primula | Ano 2018 | 91,54% |
| Primula | Ano 2019 | 86,75% |
| Primula | Ano 2020 | 69,32% |
| Anturio | Ano 2016 | 79,97% |
| Anturio | Ano 2017 | 113,83% |
| Anturio | Ano 2018 | 131,06% |
| Anturio | Ano 2019 | 115,53% |
| Anturio | Ano 2020 | 95,16% |

Para seguir com a nossa análise de cobertura vacinal será necessária a reformatação da tabela {dados} fazendo-se a transposição de suas seis colunas em três novas variáveis: **Unidade da federação, Ano e Cobertura vacinal**. Acompanhe o passo a passo abaixo:

- Criaremos três variáveis na tabela {dados}: Unidade da federação; Ano e Cobertura vacinal.
- Na operação de pivotagem vamos, então, aumentar o número de linhas e diminuir o número de colunas, deixando a tabela no formato longo. Para realizar esta operação no R, vamos utilizar a função `pivot_longer()` do pacote `tidyr`. Essa função possui três argumentos principais:
- **cols**: neste argumento especificamos quais colunas vamos transformar;
- **names_to**: neste argumento definimos o nome da variável que estamos criando e que receberá a característica que estamos trazendo do formato largo;
- **values_to**: aqui definimos o nome da variável que receberá os valores, os dados propriamente ditos.

Acompanhe os códigos abaixo e replique-os em seu RStudio:

```
# realizando a transposição dos dados do formato LARGO (*wide*)
# para uma tabela em formato LONGO (*long*)
dados |>

# Utilizando a função `pivot_longer()` para transformação de colunas
pivot_longer(

# Definindo as colunas que serão transformadas
  cols = c("Ano 2016", "Ano 2017", "Ano 2018", "Ano 2019", "Ano 2020"),

# Definindo o nome da variável nova que receberá os nomes acima
  names_to = "Ano",

# Definindo o nome da variável nova que receberá os valores da tabela
  values_to = "Cobertura Vacinal contra Hepatite B",
)
```

```
#> # A tibble: 10 × 3
#>   Municipio Ano      `Cobertura Vacinal contra Hepatite B`
#>   <chr>      <chr>      <chr>
#> 1 Primula   Ano 2016 99.55%
#> 2 Primula   Ano 2017 90.09%
#> 3 Primula   Ano 2018 91.54%
#> 4 Primula   Ano 2019 86.75%
#> 5 Primula   Ano 2020 69.32%
#> 6 Anturio   Ano 2016 79.97%
#> 7 Anturio   Ano 2017 113.83%
#> 8 Anturio   Ano 2018 131.06%
#> 9 Anturio   Ano 2019 115.53%
#> 10 Anturio  Ano 2020 95.16%
```

A tabela agora contém três colunas e dez linhas. Além disso, cada linha representa um registro específico da cobertura vacinal contra hepatite B, ano e município correspondente. Dessa forma o registro anual está concentrado em uma única coluna, e será possível a construção de um gráfico, por exemplo, de série histórica das coberturas vacinais.

Além dos argumentos citados acima, vamos precisar de mais um para atender a um detalhe da nossa transformação. Um pequeno problema surgiu ao transformar a base: toda a coluna **Ano** contém a palavra “Ano” antes de cada valor.

Para corrigir isso, vamos utilizar o argumento **name_prefix**, definindo a palavra que está repetindo na conversão da coluna para linha. Basta adicionarmos a palavra entre aspas. Veja como ficaria o comando incluindo o argumento **name_prefix** e a tabela resultante. Vamos salvar todas as modificações da tabela **{dados}** em novo objeto que chamaremos de **{dados_longos}**, replique o script abaixo em seu **RStudio**:

```
# Criando o objeto {`dados_longos`}
dados_longos <- dados |>

# Utilizando a função `pivot_longer()` para transformação de colunas
pivot_longer(

  # Definindo as colunas que serão transformadas
  cols = c("Ano 2016", "Ano 2017", "Ano 2018", "Ano 2019", "Ano 2020"),

  # Definindo o nome da variável nova que receberá os nomes acima
  names_to = "Ano",

  # Definindo o nome da variável nova que receberá os valores da tabela
  values_to = "Cobertura Vacinal contra Hepatite B",

  # Retirando a palavra "Ano " antes de cada valor da variável Ano
  # Também estamos retirando o espaço depois da palavra "Ano"
  names_prefix = "Ano "
)

# Visualizando a tabela {`dados_longos`} no formato longo
dados_longos
```



```
#> # A tibble: 10 × 3
#>   Município Ano   `Cobertura Vacinal contra Hepatite B`
#>   <chr>      <chr> <chr>
#> 1 Primula   2016  99.55%
#> 2 Primula   2017  90.09%
#> 3 Primula   2018  91.54%
#> 4 Primula   2019  86.75%
#> 5 Primula   2020  69.32%
#> 6 Anturio   2016  79.97%
#> 7 Anturio   2017 113.83%
#> 8 Anturio   2018 131.06%
#> 9 Anturio   2019 115.53%
#> 10 Anturio  2020  95.16%
```

Neste exemplo a unidade de tempo utilizada foi o ano. Mas, na rotina, várias outras medidas de tempo são comuns, como semanas epidemiológicas e a data dos primeiros sintomas. Estas duas, em particular, tendem a deixar a tabela muito “larga”, com muitas colunas (52 ou 53 colunas no caso das semanas epidemiológicas e 365 no caso dos dias) e por isso sempre as utilizamos no formato longo.

Já em outras situações pode ser necessário transformar uma tabela em formato longo para o largo. Ou seja, transpor o conteúdo armazenado em linhas para colunas. Isso pode ser útil para uma melhor visualização de uma tabela em um relatório, por exemplo. Para realizar esta operação faremos o uso da função `pivot_wider()`.

A função `pivot_wider()` é muito parecida com a `pivot_longer()` e tem os seguintes argumentos principais:

- `names_from`: argumento para especificar qual coluna será transposta;
- `values_from`: argumento para especificar qual coluna contém os valores a serem pivotados.

Vamos praticar a utilização da função `pivot_wider()`! Considere utilizar novamente a tabela que criamos anteriormente `{dados_longos}` mas, dessa vez, vamos retornar a seu formato **largo**. Observe o código abaixo e replique-o em seu **RStudio** para pivotar os dados de imunização:

```
# Transformando os dados no formato longo em formato largo
# Criando o objeto {'dados_largos'}
dados_largos <- dados_longos |>

# Utilizando a função `pivot_wider()` para transformação de colunas
pivot_wider(

# Definindo de qual variável estamos resgatando os nomes das colunas
names_from = "Ano",

# Definindo de qual variável estamos resgatando os valores das colunas
values_from = "Cobertura Vacinal contra Hepatite B")

# visualizando a tabela
dados_largos
```

```
#> # A tibble: 2 × 6
#>   Municipio `2016` `2017` `2018` `2019` `2020`
#>   <chr>      <chr> <chr>  <chr>  <chr>  <chr>
#> 1 Primula   99.55% 90.09% 91.54% 86.75% 69.32%
#> 2 Anturio   79.97% 113.83% 131.06% 115.53% 95.16%
```

Observe como a mudança na apresentação da tabela modifica a visualização dos dados. Devemos sempre realizar estas transformações para enxergar todas as variáveis do banco de dados e manipulá-las de forma eficiente. Tente utilizar esta etapa nas suas análises do dia a dia!

6.3 Separar conteúdo de variáveis em mais colunas

Você já se deparou alguma vez com exportações dos sistemas de informações que contêm variáveis com mais de um dado na mesma célula?

Isso pode ser um pouco frustrante quando tentamos tabular e analisar dados. Quando a formatação de uma base retangular não é seguida demoramos muito tempo arrumando estes dados. Além disso, pode tornar nossas operações de manipulação e análise inseguras, aumentando as chances de errar. Vamos resolver este problema. Acompanhe o exemplo abaixo.

O setor de Imunização do Estado de Rosas lhe pediu apoio para analisar os eventos adversos pós-vacinais. Para isto, enviou um banco de dados { `notificacao_eapv_2021m.xlsx` } com os registros de eventos adversos pós-vacinais com notificações realizadas no ano de 2021 no Brasil.



Lembre-se que todos os bancos de dados que utilizaremos para nossas análises encontram-se no menu lateral “Arquivos” no Ambiente Virtual deste módulo.

Rode o *script* abaixo para importar o banco de dados para manipulá-lo no R, replique o código em seu computador:

```
# Importando o banco de dados { `notificacao_eapv_2021m.xlsx` } para o `R`  
eapv_2021m <- read_xlsx('Dados/notificacao_eapv_2021m.xlsx')
```

Os profissionais de imunização selecionaram algumas variáveis relacionadas à vacina administrada para serem analisadas:

- `imunobiologico_vacina` contendo o nome da vacina,
- `dose` contendo o número da dose aplicada, e
- `data_da_aplicacao`, que armazena a data em que as respectivas doses foram aplicadas.

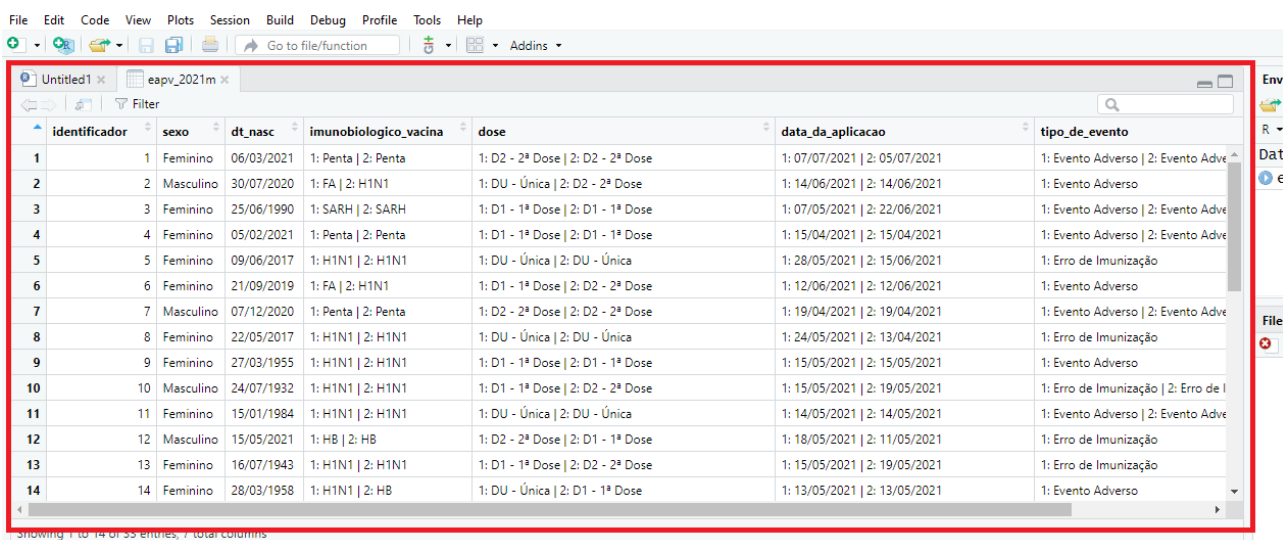
Mas, ao abrir o arquivo, você percebeu que as informações estão organizadas de uma forma que dificulta a análise, com várias informações separadas pelo símbolo (|) (barra horizontal) em uma mesma variável.

Observe esta tabela utilizando a função `View()`. Escreva o *script* abaixo em seu RStudio:

```
# Visualizando a tabela {eapv_2021m} com a função `View()`
View(eapv_2021m)
```

Observe que os dados estão dispostos como na Figura 7. Ao utilizar a função `View()` você deve ter percebido que se abriu uma nova janela no RStudio, que mostrará a tabela contida no objeto `{eapv_2021m}`. Veja:

Figura 7: Tela de visualização da tabela {eapv_2021m}.



| | identificador | sexo | dt_nasc | imunobiologico_vacina | dose | data_da_aplicacao | tipo_de_evento |
|----|---------------|-----------|------------|-----------------------|-----------------------------------|-------------------------------|---|
| 1 | 1 | Feminino | 06/03/2021 | 1: Penta 2: Penta | 1: D2 - 2ª Dose 2: D2 - 2ª Dose | 1: 07/07/2021 2: 05/07/2021 | 1: Evento Adverso 2: Evento Adverso |
| 2 | 2 | Masculino | 30/07/2020 | 1: FA 2: H1N1 | 1: DU - Única 2: D2 - 2ª Dose | 1: 14/06/2021 2: 14/06/2021 | 1: Evento Adverso |
| 3 | 3 | Feminino | 25/06/1990 | 1: SARH 2: SARH | 1: D1 - 1ª Dose 2: D1 - 1ª Dose | 1: 07/05/2021 2: 22/06/2021 | 1: Evento Adverso 2: Evento Adverso |
| 4 | 4 | Feminino | 05/02/2021 | 1: Penta 2: Penta | 1: D1 - 1ª Dose 2: D1 - 1ª Dose | 1: 15/04/2021 2: 15/04/2021 | 1: Evento Adverso 2: Evento Adverso |
| 5 | 5 | Feminino | 09/06/2017 | 1: H1N1 2: H1N1 | 1: DU - Única 2: DU - Única | 1: 28/05/2021 2: 15/06/2021 | 1: Evento de Imunização |
| 6 | 6 | Feminino | 21/09/2019 | 1: FA 2: H1N1 | 1: D1 - 1ª Dose 2: D2 - 2ª Dose | 1: 12/06/2021 2: 12/06/2021 | 1: Evento Adverso |
| 7 | 7 | Masculino | 07/12/2020 | 1: Penta 2: Penta | 1: D2 - 2ª Dose 2: D2 - 2ª Dose | 1: 19/04/2021 2: 19/04/2021 | 1: Evento Adverso 2: Evento Adverso |
| 8 | 8 | Feminino | 22/05/2017 | 1: H1N1 2: H1N1 | 1: DU - Única 2: DU - Única | 1: 24/05/2021 2: 13/04/2021 | 1: Erro de Imunização |
| 9 | 9 | Feminino | 27/03/1955 | 1: H1N1 2: H1N1 | 1: D1 - 1ª Dose 2: D1 - 1ª Dose | 1: 15/05/2021 2: 15/05/2021 | 1: Evento Adverso |
| 10 | 10 | Masculino | 24/07/1932 | 1: H1N1 2: H1N1 | 1: D1 - 1ª Dose 2: D2 - 2ª Dose | 1: 15/05/2021 2: 19/05/2021 | 1: Erro de Imunização 2: Erro de Imunização |
| 11 | 11 | Feminino | 15/01/1984 | 1: H1N1 2: H1N1 | 1: DU - Única 2: DU - Única | 1: 14/05/2021 2: 14/05/2021 | 1: Evento Adverso 2: Evento Adverso |
| 12 | 12 | Masculino | 15/05/2021 | 1: HB 2: HB | 1: D2 - 2ª Dose 2: D1 - 1ª Dose | 1: 18/05/2021 2: 11/05/2021 | 1: Erro de Imunização |
| 13 | 13 | Feminino | 16/07/1943 | 1: H1N1 2: H1N1 | 1: D1 - 1ª Dose 2: D2 - 2ª Dose | 1: 15/05/2021 2: 19/05/2021 | 1: Erro de Imunização |
| 14 | 14 | Feminino | 28/03/1958 | 1: H1N1 2: HB | 1: DU - Única 2: D1 - 1ª Dose | 1: 13/05/2021 2: 13/05/2021 | 1: Evento Adverso |

Perceba que algumas colunas possuem um ou mais dados referentes a um mesmo indivíduo separados por um caractere (|) (barra horizontal). Rode em seu **RStudio** o *script* abaixo e visualize apenas as colunas `imunobiologico_vacina`, `dose` e `data_da_aplicacao` da tabela `{eapv_2021m}`:

```
eapv_2021m |>
```

```
# Selecionando três colunas do data.frame {`eapv_2021m`}
select(imunobiologico_vacina, dose, data_da_aplicacao) |>

# Utilizando a função `head()` para visualizar as primeiras linhas da tabela
head()
```

```
#> # A tibble: 6 × 3
#>   imunobiologico_vacina dose                data_da_aplicacao
#>   <chr>                <chr>                <chr>
#> 1 1: Penta | 2: Penta  1: D2 - 2ª Dose | 2: D2 - 2ª Dose 1: 07/07/2021 | 2: 05...
#> 2 1: FA | 2: H1N1     1: DU - Única | 2: D2 - 2ª Dose  1: 14/06/2021 | 2: 14...
#> 3 1: SARH | 2: SARH   1: D1 - 1ª Dose | 2: D1 - 1ª Dose 1: 07/05/2021 | 2: 22...
#> 4 1: Penta | 2: Penta 1: D1 - 1ª Dose | 2: D1 - 1ª Dose 1: 15/04/2021 | 2: 15...
#> 5 1: H1N1 | 2: H1N1   1: DU - Única | 2: DU - Única     1: 28/05/2021 | 2: 15...
#> 6 1: FA | 2: H1N1     1: D1 - 1ª Dose | 2: D2 - 2ª Dose 1: 12/06/2021 | 2: 12...
```

Veja que para uma mesma pessoa as informações sobre diferentes doses, datas de aplicações e imunobiológicos utilizados em cada aplicação estão mesclados em uma única coluna para cada tipo de informação.

Agora, imagine que você deva utilizar as informações destas colunas para calcular se houve erro de imunização em relação ao intervalo adequado de aplicação das vacinas entre a 1ª, 2ª ou 3ª doses? Com a informação desta forma, fica muito difícil não é mesmo?

Para resolver este problema e conseguir analisar estes dados de forma adequada, devemos realizar a separação dos valores agregados em uma coluna em diversas outras variáveis. Essa operação reorganizará os valores em mais colunas, respeitando a lógica necessária para que cada célula tenha um conteúdo específico e único daquele registro.

Para esta etapa separaremos a coluna de vacina do exemplo acima, utilizando a função `separate()` do pacote `tidyr`. Essa função possui três argumentos básicos:

- `col`: indicação da coluna que será separada;
- `into`: indicação dos nomes das novas colunas a serem criadas;
- `sep`: indicação de qual separador está utilizado na célula.

Veja abaixo a aplicação da função `separate()`. Em seu `RStudio` replique o *script* abaixo:

```
eapv_2021m |>

# Dividindo a coluna `imunobiologico_vacina` em três novas colunas
separate(

  # Definindo a coluna que será separada
  col = imunobiologico_vacina,

  # Definindo os nomes das novas colunas
  into = c("vac_event_1", "vac_event_2", "vac_event_3"),

  # Definindo qual o caractere que está sendo utilizado dentro das colunas
  sep = "\\|"
) |>

# Selecionando as novas colunas para visualização
select("vac_event_1", "vac_event_2", "vac_event_3")
```

```
#> Warning: Expected 3 pieces. Missing pieces filled with `NA` in 32 rows [1, 2, 3,
#> 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].
```

```
#> # A tibble: 33 × 3
#>   vac_event_1 vac_event_2 vac_event_3
#>   <chr>      <chr>      <chr>
#> 1 "1: Penta " " 2: Penta" <NA>
#> 2 "1: FA "   " 2: H1N1" <NA>
#> 3 "1: SARH " " 2: SARH" <NA>
#> 4 "1: Penta " " 2: Penta" <NA>
#> 5 "1: H1N1 " " 2: H1N1" <NA>
#> 6 "1: FA "   " 2: H1N1" <NA>
#> 7 "1: Penta " " 2: Penta" <NA>
#> 8 "1: H1N1 " " 2: H1N1" <NA>
#> 9 "1: H1N1 " " 2: H1N1" <NA>
#> 10 "1: H1N1 " " 2: H1N1" <NA>
#> # ... with 23 more rows
```

Como resultado será armazenada uma nova tabela em que a antiga coluna `imunobiológico_vacina` foi dividida em três: `vac_event_1`, `vac_event_2` e `vac_event_3`. Além disso, o R retorna uma mensagem de *warning* avisando que algumas linhas foram preenchidas com `NA`. Isso se deve a alguns registros não estarem preenchidos com três vacinas, apenas duas. E nós definimos que criaríamos três novas colunas. Perceba que, no código acima, selecionamos as novas colunas para visualização.

Perceba que com poucas linhas de comando foi possível organizar a tabela e visualizar cada uma das doses aplicadas em uma coluna sem perder nenhuma informação. Esta etapa tornará sua análise de dados mais segura e rápida. Para separar outras colunas, repita o processo no seu `RStudio`.

Uma observação importante a ser destacada é o uso do argumento `sep`. Ele foi utilizado indicando duas barras invertidas e uma vertical `sep = "\\|"`. Isto ocorreu porque o símbolo `|` é um caractere especial, e queremos que ele seja interpretado como um caractere comum pelo uso da barra invertida (`\`). Entretanto, a própria barra invertida é também um símbolo especial, de modo que precisamos utilizar outra barra invertida novamente para que ela seja interpretada corretamente pelo R.

6.4 Recodificando linhas e colunas

Muitas vezes no processo de coleta e digitação de dados ocorrem alguns erros ortográficos que precisarão ser corrigidos antes da análise. Por exemplo, ao preencher o sexo dos pacientes atendidos em uma unidade de saúde, pode-se acabar digitando “masculinA” para um indivíduo e “masculinO” para outro, criando duas categorias diferentes que se referem ao mesmo tipo de sexo biológico. Para evitar esse tipo de erro, em uma análise adequada precisamos que as variáveis sejam recodificadas.

A recodificação em banco de dados é um procedimento frequente, pois, além de padronizar eliminando erros, possibilita a redução do tamanho do banco de dados em si, por armazenar códigos e não todo o texto da variável, além de tornar a programação de algoritmos mais eficiente. É importante frisar que para uma análise exploratória faz-se necessária a recodificação, a transformação dos códigos em dados mais informativos.

Para fazer esta recodificação utilizaremos a função `case_when()` do pacote `dplyr` no R. Esta função pode ser usada para criar variáveis a partir de variáveis existentes. Nesta função, cada “caso” é separado por vírgulas. Neste argumento, utilizamos o formato `A ~ B`, em que `A` é um código com uma proposição lógica (por exemplo, se uma variável é igual a determinado valor), e `B` indicará por qual valor iremos substituir os casos que atendem ao critério descrito na função. O operador til (`~`) faz essa comunicação.

Vejamos um exemplo de recodificação:

O profissional de vigilância necessita incluir em sua avaliação demográfica a distribuição do sexo (`CS_SEX0`) dos agravos contidos no Sinan Net. Para isto, utilizaremos o recorte do banco de dados `{NINDINET.dbf}` que chamamos de `{base_menor}`, criada anteriormente. Recodificaremos da seguinte forma:

- “M” para “Masculino”;
- “F” para “Feminino”;
- “I” para “Ignorado”, incluiremos também os registros em branco ou nulos como “Ignorado” e;
- se não existir nenhum registro no banco de dados que atenda aos critérios, um valor `NA` será atribuído.

Já vimos isso anteriormente e será um bom momento para fixarmos o conteúdo. Observe o script abaixo em que será aplicada a recodificação com atenção e replique-o em seu RStudio:

```
base_menor |>

# Renomeando os valores da variável CS_SEX0 usando a função `mutate()` e `case_when()`
mutate(
  sexo_cat = case_when(
    CS_SEX0 == "M" ~ "Masculino",
    CS_SEX0 == "F" ~ "Feminino",
    CS_SEX0 == "I" | is.na(CS_SEX0) ~ "Ignorado",
    TRUE ~ NA_character_
  )
) |>

# Visualizando a tabela {`base_menor`} recodificada
head()
```

```
#>   DT_NOTIFIC   DT_NASC CS_SEX0 CS_RACA ID_MN_RESI ID_AGRAVO  sexo_cat
#> 1 2012-04-11 2012-04-04      M      4    610213    A509 Masculino
#> 2 2010-09-17 1988-04-23      M      1    610213     W64 Masculino
#> 3 2010-10-19 1971-03-25      M     NA    610250     X58 Masculino
#> 4 2008-04-14 1928-05-29      F      4    610213     A90 Feminino
#> 5 2011-06-20 2002-09-18      M      4    610250     B19 Masculino
#> 6 2008-02-12 1953-08-01      F      9    610213     A90 Feminino
```

Observe em seu *output* como a função `case_when()` criou os valores para a nova coluna:

- `CS_SEX0 == "M" ~ "Masculino"`: Se o valor na coluna `CS_SEX0` for "M", então o valor na coluna deve ser "Masculino";
- `CS_SEX0 == "F" ~ "Feminino"`: Se o valor na coluna `CS_SEX0` for "F", então o valor na coluna deve ser "Feminino";
- `CS_SEX0 == "I" | is.na(CS_SEX0) ~ "Ignorado"`: Se o valor na coluna `CS_SEX0` for "I" **OU** nulo, o valor na coluna deve ser "Ignorado".
- `TRUE ~ NA_character_`: Se nenhum dos critérios anteriores forem atendidos, o valor da nova coluna deverá ser NA. Como a nova variável criada será do tipo `character`, o NA pode também ser desse tipo.

Agora, repetiremos a mesma operação realizada para a variável epidemiológica Raça/cor (`CS_RACA`), veja mais uma vez:

```
base_menor |>
```

```
# Renomeando os valores da variável CS_RACA usando a função `mutate()` e `case_when()`  
mutate(  
  raca_cor_cat = case_when(  
  
    # Se o valor da coluna for igual a "1" transforme para "Branca"  
    CS_RACA == "1" ~ "Branca",  
  
    # Se o valor da coluna for igual a "2" transforme para "Preta"  
    CS_RACA == "2" ~ "Preta",  
  
    # Se o valor da coluna for igual a "3" transforme para "Amarela"  
    CS_RACA == "3" ~ "Amarela",  
  
    # Se o valor da coluna for igual a "4" transforme para "Parda"  
    CS_RACA == "4" ~ "Parda",  
  
    # Se o valor da coluna for igual a "5" transforme para "Indígena"  
    CS_RACA == "5" ~ "Indígena",  
  
    # Se o valor da coluna for igual a "9" ou nulo transforme para "Ignorado"  
    CS_RACA == "9" | is.na(CS_RACA) ~ "Ignorado",  
  
    # Caso acontecer um valor diferente dos citados acima, transforme para "NA"  
    TRUE ~ NA_character_  
  )  
) |>  
  
# Utilizando a função `head()` para visualizar as primeiras linhas  
head()
```

```
#>   DT_NOTIFIC   DT_NASC CS_SEXO CS_RACA ID_MN_RESI ID_AGRAVO  raca_cor_cat
#> 1 2012-04-11 2012-04-04      M      4    610213    A509      Parda
#> 2 2010-09-17 1988-04-23      M      1    610213     W64      Branca
#> 3 2010-10-19 1971-03-25      M     NA    610250     X58      Ignorado
#> 4 2008-04-14 1928-05-29      F      4    610213     A90      Parda
#> 5 2011-06-20 2002-09-18      M      4    610250     B19      Parda
#> 6 2008-02-12 1953-08-01      F      9    610213     A90      Ignorado
```

Observe que as variáveis foram todas recodificadas, facilitando a identificação do conteúdo da coluna sem uso do dicionário de dados.

Outra organização nos dados que o profissional de vigilância realiza com frequência para suas análises é a criação de categorias, como a transformação das idades em faixas etárias para construir análises comparativas entre grupos etários diferentes ou representação gráfica como pirâmides etárias. Para isso, a categorização é um procedimento que recodifica uma variável em tipo de dado diferente, pois está transformando um valor do tipo número em um valor do tipo texto.

Para fazer esta categorização utilizaremos aqui a função `if_else()` do pacote `dplyr` no R. Esta função cria uma coluna baseada a um critério específico, equivalente às sentenças lógicas do tipo `SE` do Microsoft Excel. Veja o exemplo abaixo.

Ainda utilizando a tabela `{base}`, que está armazenando os dados importados do banco de dados `{NINDINET.dbf}`, o profissional de vigilância necessitará analisar a distribuição por faixa etária de todos os agravos notificados no Sinan Net.

Para facilitar a visualização vamos selecionar algumas colunas que alteramos. Calma que logo abaixo explicamos os códigos para você!

Insira os comandos do *script* abaixo no seu `RStudio` e observe como ficaram as colunas modificadas:

```
base |>
```

```
# Utilizando a função `mutate()` para criar colunas
mutate(

  # Criando uma coluna de idade conforme a codificação da variável NU_IDADE_N
  idade_anos = if_else(str_sub(NU_IDADE_N, 1, 1) == "4",
as.numeric(str_sub(NU_IDADE_N, 2, 4)), 0),

  # Criando uma coluna de faixa etária a partir da variável idade dos casos notificados
  # utilizando a função `cut()`
  fx_etaria = cut(
    # Definindo qual variável será classificada em faixas
    x = idade_anos,

    # Definindo os pontos de corte das classes
    breaks = c(0, 10, 20, 60, Inf),

    # Definindo os rótulos das classes
    labels = c("0-9 anos", "10-19 anos", "20-59 anos", "60 anos e+"),

    # Definindo o tipo do ponto de corte
    right = FALSE
  )
) |>

# Selecionando as variáveis que queremos utilizar com a função `select()`
select(NU_NOTIFIC, ID_AGRAVO, NU_IDADE_N, idade_anos, fx_etaria) |>

# Visualizando a tabela {`base`} modificada
head()
```

```
#>   NU_NOTIFIC ID_AGRAVO NU_IDADE_N idade_anos fx_etaria
#> 1    7671320    A509      2001         0  0-9 anos
#> 2    0855803    W64      4022        22 20-59 anos
#> 3    8454645    X58      4039        39 20-59 anos
#> 4    3282723    A90      4079        79 60 anos e+
#> 5    9799526    B19      4008         8  0-9 anos
#> 6    7275624    A90      4054        54 20-59 anos
```

O código acima possui vários detalhes que são necessários explicar. Perceba que estamos utilizando a função `mutate()` para criar duas colunas: `idades_anos` e `fx_etaria`. Na nova variável `idade_anos` usamos a função `if_else()`, que verifica se o código da idade no banco de dados é "4" (que significa que o registro da idade é em anos).

Isso é necessário pois o primeiro dígito da variável `idade` no banco do Sinan Net informa se a idade está em anos, meses, dias ou horas. Se o critério for verdadeiro, a função `str_sub()` do pacote `stringr` extrai os dois últimos dígitos, pegando somente o valor da idade. Se o critério não for verdadeiro, será registrado o valor zero. A função `as.numeric()` transforma o resultado para o tipo numérico.

Para criar a variável `fx_etaria`, estamos utilizando a função `cut()`. Essa função classifica a variável recém-criada `idade_anos` em quatro categorias. Os argumentos utilizados pela função são:

- **x**: variável numérica que será categorizada;
- **breaks**: cortes de idades. Aqui definimos os cortes começando em 0, depois em 10, 20, 60 e, o último, incluirá qualquer valor depois de 60 (`Inf`, que denota infinito);
- **labels**: rótulos para as classes;
- **right**: define se corte da categoria acontece antes ou depois dos números definidos em **breaks**. Por exemplo, com `right = TRUE`, uma idade de 20 anos será incluída na categoria 10-19 anos. Se `right = FALSE`, a idade de 20 anos será incluída na categoria 20-59 anos.

Pronto. Perceba que agora temos os casos notificados no Sinan Net organizados por faixa etária na variável `fx_etaria`, nas seguintes categorias: "0-9 anos", "10-19", "20-59 anos", "60 anos e+". Após esta categorização você será capaz de perceber se existem diferença entre os agravos em relação à idade.

Agora que você já sabe reorganizar suas tabelas, corrigir erros de codificação, recategorizá-las e armazenar corretamente seus dados, vamos cruzar tabelas para construir sua análise de situação de saúde.

7. Unindo tabelas com o R

Já vimos nos tópicos anteriores que os dados de vigilância em saúde pública podem ser provenientes de diversas fontes, tais como buscas em prontuários, laboratórios públicos ou privados, sistemas de informação diversos, etc. O profissional de vigilância em seu dia a dia deve então consultar diversas fontes de dados para a construção de sua rotina de avaliação dos agravos, doenças e eventos de saúde pública.

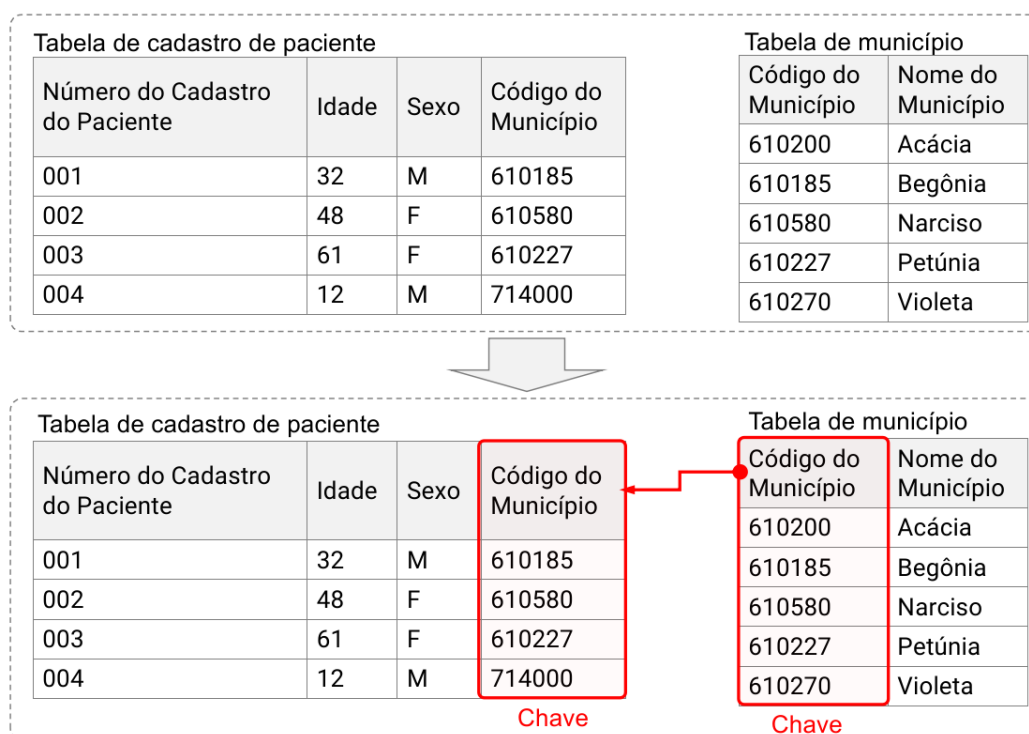
É comum que para realizar uma investigação você necessite consultar os resultados de exames laboratoriais ou o cadastro de endereço de um paciente para concluir sua notificação garantindo a qualidade e veracidade de seus dados. Por vezes, é necessário unir planilhas ou relacioná-las para trabalhar em conjunto dando mais velocidade a este tipo de ação.

Muitas vezes na vigilância cruzamos a tabela Classificação Brasileira de Ocupações (CBO) fornecida pelo Datasus com os códigos de profissão notificados no Sinan NET, ou mesmo a tabela de CID-10 com as ocorrências de óbito (SIM - Sistema de Informações de Óbitos) do seu município. Este é um trabalho exaustivo e para alguns agravos se torna impossível, como durante a pandemia de Covid-19 em que o volume de notificações é enorme. Neste tópico iremos aprender algumas das ações mais comuns de relacionamento entre bancos de dados.

A união ou junção de bancos de dados é um procedimento realizado sempre entre duas ou mais bases. Para essa ação é necessário que os bancos de dados que serão unidos contenham variáveis que identifiquem unicamente cada registro (linha) para que possam ser usadas para conectá-las. Essas variáveis são chamadas chaves (em inglês, *keys*). Outra importante necessidade é que estas variáveis possuam o mesmo tipo de dados em ambas as tabelas (numérico, texto, etc).

Em junções simples de bases, basta uma única chave para se identificar o registro e seguir com a união conforme a Figura 8.

**Figura 8: Relação de união entre a tabelas de
cadastro de pacientes e a tabela de municípios**



Perceba que temos duas tabelas: a da esquerda é o cadastro de paciente e a da direita o cadastro de municípios no Estado de Rosas. Na primeira tabela, além de dados pessoais dos pacientes, há o código do município onde o paciente reside, mas não há o nome do município. Na tabela da direita há o código e o nome do município, mas não há dados dos pacientes. Neste caso podemos relacioná-las utilizando a chave (*key*) em comum: o **Código do Município**.

Dessa forma, a tabela de paciente se conecta a de município por uma única variável, a chave **Código do município**. O resultado desse processo acrescenta mais uma coluna com o nome do município, como pode ser visto na Figura 9.

Figura 9: Tabela unificada entre a tabela de cadastro de pacientes e de municípios

Tabela unificada

| Número do Cadastro do Paciente | Idade | Sexo | Código do Município | Município |
|--------------------------------|-------|------|---------------------|-----------|
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |
| 004 | 12 | M | 714000 | NA |

Agora veremos a seguir quais são as outras possibilidades de relacionamento entre tabelas que poderemos utilizar com apoio da linguagem **R** e os produtos resultantes das uniões.

7.1 Tipos de cruzamento de dados (Join)

No exemplo anterior, percebemos que, após a união, a tabela de pacientes, localizada à esquerda, recebeu uma nova coluna e manteve todos os seus registros, embora um registro não tenha sido encontrado na tabela de municípios. Já na tabela de municípios, à direita, restaram aqueles em comum com a tabela de pacientes. Nesta ação utilizamos a função *join* do inglês, uma operação de junção que combina colunas de uma ou mais tabelas em um banco de dados relacional. Agora, vamos aprender os tipos de *join* presentes no pacote **dplyr**.

Observe o procedimento de união das duas tabelas apresentadas na Figura 10. A tabela de cadastros de paciente, que chamaremos de **tabela_a**, e a tabela de municípios, que chamaremos de **tabela_b**, serão unidas utilizando a chave **cod_mun**. Vamos destacar os principais tipos de união, suas descrições detalhando o resultado dos procedimentos e um exemplo de uso da função correspondente no **dplyr**.

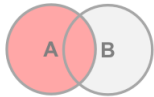
Figura 10: União de tabelas.

| Cadastro de paciente (tabela_a) | | | | | Cadastro de municípios (tabela_b) | | |
|---------------------------------|-------|------|---------|---|-----------------------------------|----------|---|
| id_cadastro | idade | sexo | cod_mun | | cod_mun | nome_mun | |
| 001 | 32 | M | 610185 | + | 610200 | Acácia | = |
| 002 | 48 | F | 610580 | | 610185 | Begônia | |
| 003 | 61 | F | 610227 | | 610580 | Narciso | |
| 004 | 12 | M | 714000 | | 610227 | Petúnia | |
| | | | | | 610270 | Violeta | |

**Tipo de
união**

Descrição

Função do dplyr e exemplo de uso

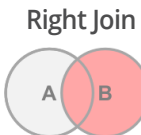


Todos os registros da tabela **A** são mantidos e os registros da tabela **B** apenas se estiverem correspondência em **A**. Caso não houver correspondência, é adicionado NA.

`left_join(tabela_a, tabela_b, by = "cod_mun")`

| id_cadastro | idade | sexo | cod_mun | nome_mun |
|-------------|-------|------|---------|----------|
| | | | 610200 | Acácia |
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |
| 004 | 12 | M | 714000 | |
| | | | 610270 | Violeta |

| id_cadastro | idade | sexo | cod_mun | nome_mun |
|-------------|-------|------|---------|----------|
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |
| 004 | 12 | M | 714000 | NA |

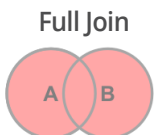


Todos os registros da tabela **B** são mantidos e os registros da tabela **A** apenas se estiverem correspondência em **B**. Caso não houver correspondência, é adicionado NA.

`right_join(tabela_a, tabela_b, by = "cod_mun")`

| id_cadastro | idade | sexo | cod_mun | nome_mun |
|-------------|-------|------|---------|----------|
| | | | 610200 | Acácia |
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |
| 004 | 12 | M | 714000 | |
| | | | 610270 | Violeta |

| id_cadastro | idade | sexo | cod_mun | nome_mun |
|-------------|-------|------|---------|----------|
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |
| NA | NA | NA | 610200 | Acácia |
| NA | NA | NA | 610270 | Violeta |

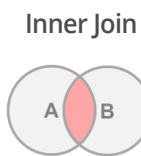


Todos os registros da tabela **A** e da tabela **B** são mantidos. Nos registros sem correspondência é adicionado NA.

`full_join(tabela_a, tabela_b, by = "cod_mun")`

| id_cadastro | idade | sexo | cod_mun | nome_mun |
|-------------|-------|------|---------|----------|
| | | | 610200 | Acácia |
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |
| 004 | 12 | M | 714000 | |
| | | | 610270 | Violeta |

| id_cadastro | idade | sexo | cod_mun | nome_mun |
|-------------|-------|------|---------|----------|
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |
| 004 | 12 | M | 714000 | NA |
| NA | NA | NA | 610200 | Acácia |
| NA | NA | NA | 610270 | Violeta |

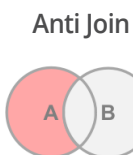


Somente os registros com correspondência em ambas tabelas são mantidos.

`inner_join(tabela_a, tabela_b, by = "cod_mun")`

| id_cadastro | idade | sexo | cod_mun | nome_mun |
|-------------|-------|------|---------|----------|
| | | | 610200 | Acácia |
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |
| 004 | 12 | M | 714000 | |
| | | | 610270 | Violeta |

| id_cadastro | idade | sexo | cod_mun | nome_mun |
|-------------|-------|------|---------|----------|
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |

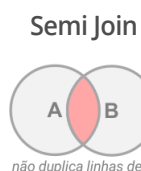


Somente os registros da tabela **A** que **NÃO** tem correspondência na tabela **B** são mantidos.

`anti_join(tabela_a, tabela_b, by = "cod_mun")`

| id_cadastro | idade | sexo | cod_mun | nome_mun |
|-------------|-------|------|---------|----------|
| | | | 610200 | Acácia |
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |
| 004 | 12 | M | 714000 | |
| | | | 610270 | Violeta |

| id_cadastro | idade | sexo | cod_mun |
|-------------|-------|------|---------|
| 004 | 12 | M | 714000 |



Somente os registros da tabela **A** que tem correspondência na tabela **B** são mantidos. Não há duplicidade de linhas da tabela **A** com essa união.

`semi_join(tabela_a, tabela_b, by = "cod_mun")`

| id_cadastro | idade | sexo | cod_mun | nome_mun |
|-------------|-------|------|---------|----------|
| | | | 610200 | Acácia |
| 001 | 32 | M | 610185 | Begônia |
| 002 | 48 | F | 610580 | Narciso |
| 003 | 61 | F | 610227 | Petúnia |
| 004 | 12 | M | 714000 | |
| | | | 610270 | Violeta |

| id_cadastro | idade | sexo | cod_mun |
|-------------|-------|------|---------|
| 001 | 32 | M | 610185 |
| 002 | 48 | F | 610580 |
| 003 | 61 | F | 610227 |

Lembre-se que as funções *join* possuem os mesmos argumentos, sendo os principais:

- os nomes das tabelas a serem unidas, e
- a indicação da variável chave no argumento `by`.

Agora iremos cruzar os nomes dos municípios contidos no banco de dados `{tabela_municipios.xlsx}` com os códigos de municípios notificando agravos no Estado de Rosas por meio do Sinan Net `{NINDINET.dbf}`.

Para isto, realizaremos o cruzamento do objeto `{base_menor}`, criado a partir do banco de dados `{NINDINET.dbf}`, com a tabela de municípios `{tabela_municipios.xlsx}`, exportada do IBGE, disponível no menu lateral “Arquivos”, no Ambiente Virtual deste curso.

Vamos lá, primeiro vamos importar o banco de dados com nome dos municípios de Rosas e seus respectivos códigos. Acompanhe os *scripts* abaixo e replique-o em seu RStudio.

```
# Importando o banco de dados {`tabela_municipios.xlsx`} para o `R`  
tabela_municipios <- read_excel("Dados/tabela_municipios.xlsx",  
                                sheet = 1,  
                                skip = 0)
```

A seguir, iremos realizar a união das tabelas considerando que as duas possuem uma variável em comum: o código de identificação do município. Para `{base_menor}` esta variável possui o nome `ID_MN_RESI`, enquanto para `{tabela_municipios}` esta variável possui o nome `ID_MUNICIPIO`. Antes de fazermos a união, precisamos checar se são compatíveis, isto é, se são o mesmo tipo de variável.

Acompanhe o *script* abaixo e replique-o em seu computador:

```
# Verificando o tipo de variável na coluna `ID_MN_RESI` do NINDINET  
class(base_menor$ID_MN_RESI)
```

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

```
# Verificando o tipo de variável na coluna `ID_MUNICIPIO` da tabela de municípios do IBGE  
class(tabela_municipios$ID_MUNICIPIO)
```

```
#> [1] "character"
```

Perceba no output que na `{base_menor}` a coluna com os códigos de municípios é um número inteiro (`integer`) e que para o objeto `{tabela_municipios}` obtemos uma variável em formato de texto (`character`).

Para evitar erros de compatibilidade, vamos transformar a coluna `ID_MUNICIPIO` de `{tabela_municipios}` também em uma variável numérica. Observe os códigos abaixo, e replique-os em seu RStudio:

```
# Transformando apenas a variável `ID_MUNICIPIO` do data.frame `{tabela_municipios}`  
# e utilizando a função `as.integer()` para torná-la do tipo numérica  
tabela_municipios$ID_MUNICIPIO <- as.integer(tabela_municipios$ID_MUNICIPIO)
```

Pronto! Agora, já é possível realizar a união das tabelas utilizando as colunas com os códigos dos municípios de forma segura. Continue acompanhando o *script* abaixo e replique-o em seu RStudio:

```
left_join(  
  
  # Unindo a tabela `{base_menor}`  
  x = base_menor,  
  
  # com a tabela `{tabela_municipios}`  
  y = tabela_municipios,  
  
  # Selecionando as colunas `ID_MN_RESI` e `ID_MUNICIPIO` para unir os bancos de dados  
  by = c("ID_MN_RESI" = "ID_MUNICIPIO")) |>  
  
  # Visualizando a união realizada  
  head()
```

```
#>   DT_NOTIFIC   DT_NASC CS_SEXO CS_RACA ID_MN_RESI ID_AGRAVO ID_UF NM_ESTADO
#> 1 2012-04-11 2012-04-04      M      4    610213    A509    33    Rosas
#> 2 2010-09-17 1988-04-23      M      1    610213     W64    33    Rosas
#> 3 2010-10-19 1971-03-25      M     NA    610250     X58    33    Rosas
#> 4 2008-04-14 1928-05-29      F      4    610213     A90    33    Rosas
#> 5 2011-06-20 2002-09-18      M      4    610250     B19    33    Rosas
#> 6 2008-02-12 1953-08-01      F      9    610213     A90    33    Rosas
#>   NM_MUNICIPIO
#> 1      Prímula
#> 2      Prímula
#> 3      Papoila
#> 4      Prímula
#> 5      Papoila
#> 6      Prímula
```

Pronto. Agora temos uma tabela {base_menor} com a inclusão dos nomes de todos os municípios que tiveram pessoas notificadas.

8. Transformando e limpando textos

Para construir a sua rotina automatizada de análises de dados, o profissional de vigilância deverá aprender como trabalhar com textos dentro do R. Já vimos que a linguagem de programação R é uma poderosa ferramenta para a preparação e a limpeza de dados. Em programação, informações de texto são denominadas *strings* (termo em inglês para “cadeia de caracteres”). Uma *string* não é nada mais do que uma sequência de caracteres que podem ser utilizados como informação em um programa. Conforme vimos no módulo anterior, *strings* de texto no R são representados por um conjunto de caracteres entre duas aspas.

8.1 Contando caracteres e extraindo textos

O profissional de vigilância em saúde precisará seguir com a sua análise sobre os agravos de notificação. Para isso iremos continuar a explorar a tabela de classificação de agravos CID-10 {CID-10-CATEGORIAS.CSV} exportada do Datasus. Lembramos que esse banco de dados está disponível no menu lateral “Arquivos” no Ambiente Virtual do módulo.

Considere corrigir os textos digitados errados. Isto ocorre muito em campos abertos de formulários na área da saúde onde cada profissional escreve da maneira que deseja as palavras, deixando a coluna de difícil interpretação.

Por exemplo, você já deve ter se deparado com alguma variável **NOME**, contendo caracteres de nome com texto escrito de diferentes modos: “Fernanda Lima”, “Fernnda Lima”, *Fenanda Lima* ou “Fernanda Llima”. Para estes casos, o R possui um pacote chamado **stringr** do **tidyverse** com funções para a manipulação de texto ou *strings* de uma forma mais simples e intuitiva, e nos apoiará na correção deste tipo de problema.



Atenção

Para seguir com todas as análises no R, os pacotes devem estar instalados e carregados. Já fizemos esta etapa no início do Módulo. Caso tenha reiniciado seus estudos, vá até o começo deste módulo para instalar e carregar novamente os pacotes que utilizados nesta sessão.

Vamos lá! Acompanhe todos os *scripts* abaixo e replique-os em seu RStudio.

Primeiramente, importaremos a base de dados para o R utilizando a função `read_csv2()`.

```
# Importando o banco de dados { `CID-10-CATEGORIAS.CSV` } para o `R`  
CID10 <- read_csv2("Dados/CID-10-CATEGORIAS.CSV",  
                  locale = locale(encoding = "latin1"))
```

Em seguida, iremos selecionar as 15 primeiras linhas com valores utilizando a função `slice()` e realizar uma transformação para um conjunto simples de texto, utilizando a função `pull()`.

```
# Criando um novo objeto com as seleções feitas
nomes_cid <- CID10 |>

# Selecionar apenas as linhas de interesse (linhas 1 a 15)
slice(n = 1:15) |>

# Selecionando apenas a variável "DESCRICA0" e transformando em `character` simples
pull(DESCRICA0)
```

Com este exemplo, percorreremos nesta subseção alguns dos tratamentos mais utilizados com textos.

Perceba que ao executarmos os comandos acima, retornamos na tela do **RStudio** (*output*) dentro do objeto `nomes_cid` as 15 primeiras linhas da coluna `DESCRICA0`. Agora utilizaremos esse objeto para obter algumas informações sobre as variáveis de forma a padronizá-las para nossa análise. Esta etapa é crucial para tornar nossa avaliação mais assertiva!

Agora, considere que necessitamos saber quantos caracteres existem em cada linha. No **R** utilizaremos a função `str_length()` para esta avaliação. O primeiro argumento da função que deverá ser preenchido é um vetor contendo variáveis de texto (*strings*).

Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Contando quantos caracteres existem em cada linha
str_length(nomes_cid)
```

```
#> [1] 6 28 31 10 40 77 8 43 59 57 70 81 30 28 18
```

Observe que neste caso, a função retornou um vetor contendo o número de caracteres de cada *string* do vetor utilizado no *output*. Assim, temos linhas que contêm 6 caracteres, outros com 28, outros com 31 caracteres, e assim por diante.

Considere agora que o profissional de vigilância de Rosas necessita extrair apenas o início de cada doença, e descartar o restante. É possível utilizar a função `str_sub(string, start, end)`, utilizando seus três argumentos:

1. `string`: o vetor de texto,
2. `start`: a posição de início de extração do texto,
3. `end`: a posição de fim de extração do texto.

Acompanhe o script abaixo em que solicitamos ao R que extraia caracteres entre a 1ª até a 11ª posição na linha da tabela `{nomes_cid}`. Replique o código em seu RStudio:

```
# Extraindo caracteres entre a 1ª e a 11ª posição da linha  
str_sub(nomes_cid, start = 1, end = 11)
```

```
#> [1] "Cólera"      "Febres tifó" "Outras infe" "Shigelose"  "Outras infe"  
#> [6] "Outras into" "Amebíase"    "Outras doen" "Infecções i" "Diarréia e "  
#> [11] "Tuberculose" "Tuberculose" "Tuberculose" "Tuberculose" "Tuberculose"
```

Observe que obtivemos no *output* todos os *strings* cortados entre o 1º e 11º caracteres da linha da variável `nomes_cid`. Mas perceba que para padronizar as doenças ainda será necessário refinar ainda mais esta extração, de forma a tornar a variável padronizada. Continue o curso para saber mais.

Esta função também é bastante flexível, e você pode definir posições de início de fim para cada elemento da linha ou vetor de texto apontando o lugar exato que deseja extrair os caracteres.

Para o próximo exemplo a posição de início da extração será o primeiro caracter do texto, e a posição de final da extração será o número de caracteres da primeira palavra de cada elemento de nosso vetor, que foi determinado manualmente.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Indicando a posição final da primeira palavra de cada texto  
str_sub(nomes_cid,  
        start = 1,  
        end = c(6,6,6,10,6,6,8,6,9,8,11,11,11,11,11))
```

```
#> [1] "Cólera"      "Febres"      "Outras"      "Shigelose"   "Outras"  
#> [6] "Outras"      "Amebíase"    "Outras"      "Infecções"   "Diarréia"  
#> [11] "Tuberculose" "Tuberculose" "Tuberculose" "Tuberculose" "Tuberculose"
```

Veja como o resultado foi mais preciso na padronização das palavras, as quais neste caso são doenças. Foi fácil, não é mesmo?! Treine com o banco de dados da sua vigilância.



O pacote `stringr` do `tidyverse` possui todas as suas funções iniciadas com `str_`. Isso tornará mais fácil carregá-las quando necessário. Pratique!

8.2 Alterando a ordem e reordenando texto

Na maioria das vezes em que filtramos uma planilha ou tabela, quase sempre organizamos uma variável em ordem alfabética. Já vimos que a função `arrange()` cumpre este objetivo, mas com o pacote `stringr`, você também poderá reordenar um vetor de *strings*. Para isso, bastaria utilizar a função `str_sort()`.

Acompanhe o *script* abaixo e replique-o em seu `RStudio`:

```
# Ordenando o objeto {'nomes_cid'} para o formato ascendente (A-Z)
str_sort(nomes_cid)
```

```
#> [1] "Amebíase"
#> [2] "Cólera"
#> [3] "Diarréia e gastroenterite de origem infecciosa presumível"
#> [4] "Febres tifóide e paratifóide"
#> [5] "Infecções intestinais virais, outras e as não especificadas"
#> [6] "Outras doenças intestinais por protozoários"
#> [7] "Outras infecções intestinais bacterianas"
#> [8] "Outras infecções por Salmonella"
#> [9] "Outras intoxicações alimentares bacterianas, não classificadas em outra parte"
#> [10] "Shigelose"
#> [11] "Tuberculose das vias respiratórias, sem confirmação bacteriológica ou histológica"
#> [12] "Tuberculose de outros órgãos"
#> [13] "Tuberculose do sistema nervoso"
#> [14] "Tuberculose miliar"
#> [15] "Tuberculose respiratória, com confirmação bacteriológica e histológica"
```

Agora, vamos ordenar o texto de forma decrescente (Z-A). Esta ação será possível ao adicionarmos o argumento `decreasing = TRUE`.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Ordenando o objeto {`nomes_cid`} para o formato decrescente (Z-A)  
str_sort(nomes_cid, decreasing = TRUE)
```

```
#> [1] "Tuberculose respiratória, com confirmação bacteriológica e histológica"  
#> [2] "Tuberculose miliar"  
#> [3] "Tuberculose do sistema nervoso"  
#> [4] "Tuberculose de outros órgãos"  
#> [5] "Tuberculose das vias respiratórias, sem confirmação bacteriológica ou histológica"  
#> [6] "Shigelose"  
#> [7] "Outras intoxicações alimentares bacterianas, não classificadas em outra parte"  
#> [8] "Outras infecções por Salmonella"  
#> [9] "Outras infecções intestinais bacterianas"  
#> [10] "Outras doenças intestinais por protozoários"  
#> [11] "Infecções intestinais virais, outras e as não especificadas"  
#> [12] "Febres tifóide e paratifóide"  
#> [13] "Diarréia e gastroenterite de origem infecciosa presumível"  
#> [14] "Cólera"  
#> [15] "Amebíase"
```

Observe em seu *output* que a palavra “Amebíase” passou da primeira posição para a última em apenas uma linha de comando `str_sort(nomes_cid, decreasing = TRUE)`.

Agora vamos aprender a unir palavras e outros textos!

8.3 Unindo textos

Com a linguagem R também será possível unir um conjunto de palavras em um único *string*. Para esta ação iremos utilizar a função `str_c()`, do pacote `stringr`.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Unindo *strings* para a formação da palavra "Tuberculose"  
str_c("Tuber", "cuLo", "se")
```

```
#> [1] "TubercuLose"
```

Observe que todos os *strings* são unidos sem qualquer tipo de separação entre si. Esta é a definição padrão da função `str_c()`. Mas, e se quiséssemos definir que queremos que cada string seja separado por um espaço?

Para responder a esta pergunta adicionaremos o argumento `sep = " "` (igual à vazio). E ainda utilizaremos como argumento um conjunto de palavras, formando uma expressão no final.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Unindo *strings* incluindo espaço entre elas  
# adicionaremos o argumento `sep = " "`  
str_c("Tuberculose", "respiratória", "com", "confirmação",  
      "bacteriológica", "e histológica", sep = " ")
```

```
#> [1] "Tuberculose respiratória, com confirmação bacteriológica e histológica"
```

Perceba que nos dois exemplos que praticamos cada string é o próprio argumento da função, mas necessitam estar separados por uma vírgula.

Lembre-se que é uma boa prática quando utilizamos uma linguagem de programação salvar as modificações realizadas em um novo objeto. Neste caso ele será do tipo vetor e chamaremos de `{agravo}`.

Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Salvando as modificações no objeto {`agravo`}
agravo <- c("Tuberculose", "respiratória", "com", "confirmação",
           "bacteriológica", "e histológica")
```

É possível com o R unirmos o vetor em um único *string* de texto. Para isso devemos utilizar o argumento `collapse()` e precisamos definir novamente uma separação por espaço, indicando por um espaço entre duas aspas (" "). Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Unindo *strings* armazenadas no objeto {`agravo`}
# em um único *string* de texto
str_c(agravo, collapse = " ")
```

```
#> [1] "Tuberculose respiratória, com confirmação bacteriológica e histológica"
```

Pronto! Agora você já consegue separar e unir textos ou palavras da forma necessária para suas análises. Pratique!



Imagine que você possui uma lista de nomes de pacientes de uma UBS e necessita listar apenas os primeiros nomes de cada um deles. Como você faria?

Basta utilizar a função `word()` do pacote `stringr`. Acompanhe o script de exemplo abaixo e replique-o em seu RStudio:

```
# Criando um vetor de nomes
nomes <- c(
  "Caroline Nogueira",
  "Cauã Vieira Gomes",
  "Leandro Mendes",
  "Luigi Aragão",
  "Isabella da Rocha",
  "Luiza Rezende Silva Lopes"
)

# Extraíndo o primeiro nome
word(nomes, 1)
```

```
#> [1] "Caroline" "Cauã"      "Leandro"  "Luigi"
      "Isabella" "Luiza"
```

8.4 Transformando textos

Agora que já aprendemos a importância das etapas de extração, ordenação, junção e separação de textos para o tratamento e transformação dos dados, vamos ampliar mais um pouco as possibilidades de uso do pacote `stringr`.

Com o pacote `stringr` podemos realizar operações que modificam os caracteres facilmente, como por exemplo, transformar letras maiúsculas em minúsculas, ou formato de *título*, ou formato de *frase*. O formato de *título* ocorre quando queremos que apenas a primeira letra de cada palavra esteja como maiúscula, enquanto o formato de *frase* transforma apenas a primeira letra em maiúscula, enquanto as restantes são minúsculas.

Estas ações são fundamentais quando estamos preparando nossos dados para serem analisados tornando nossa análise segura e rápida. Para isso, utilizamos respectivamente as funções `str_to_upper()`, `str_to_lower()`, `str_to_title()`, e `str_to_sentence()`.

Para apresentação deste pacote iremos utilizar o objeto criado no tópico anterior: `nomes_cid`. Primeiro, iremos transformar todos as letras em maiúsculas, utilizando a função `str_to_upper()`.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Transformando todos as letras em maiúsculas  
str_to_upper(nomes_cid)
```

```
#> [1] "CÓLERA"  
#> [2] "FEBRES TIFÓIDE E PARATIFÓIDE"  
#> [3] "OUTRAS INFECÇÕES POR SALMONELLA"  
#> [4] "SHIGUELOSE"  
#> [5] "OUTRAS INFECÇÕES INTESTINAIS BACTERIANAS"  
#> [6] "OUTRAS INTOXICAÇÕES ALIMENTARES BACTERIANAS, NÃO CLASSIFICADAS EM OUTRA PARTE"  
#> [7] "AMEBÍASE"  
#> [8] "OUTRAS DOENÇAS INTESTINAIS POR PROTOZOÁRIOS"  
#> [9] "INFECÇÕES INTESTINAIS VIRAIS, OUTRAS E AS NÃO ESPECIFICADAS"  
#> [10] "DIARRÉIA E GASTROENTERITE DE ORIGEM INFECCIOSA PRESUMÍVEL"  
#> [11] "TUBERCULOSE RESPIRATÓRIA, COM CONFIRMAÇÃO BACTERIOLÓGICA E HISTOLÓGICA"  
#> [12] "TUBERCULOSE DAS VIAS RESPIRATÓRIAS, SEM CONFIRMAÇÃO BACTERIOLÓGICA OU HISTOLÓGICA"  
#> [13] "TUBERCULOSE DO SISTEMA NERVOSO"  
#> [14] "TUBERCULOSE DE OUTROS ÓRGÃOS"  
#> [15] "TUBERCULOSE MILIAR"
```

Como resultado, você obtém todos os caracteres de texto de seu vetor em letras maiúsculas.

Agora, vamos transformar todos as letras em minúsculas, utilizando a função `str_to_lower()`. Acompanhe o script abaixo e replique-o em seu **RStudio**:

```
# Transformando todos as letras em minúsculas  
str_to_lower(nomes_cid)
```

```
#> [1] "cólera"  
#> [2] "febres tifóide e paratifóide"  
#> [3] "outras infecções por salmonella"  
#> [4] "shigelose"  
#> [5] "outras infecções intestinais bacterianas"  
#> [6] "outras intoxicações alimentares bacterianas, não classificadas em outra parte"  
#> [7] "amebíase"  
#> [8] "outras doenças intestinais por protozoários"  
#> [9] "infecções intestinais virais, outras e as não especificadas"  
#> [10] "diarréia e gastroenterite de origem infecciosa presumível"  
#> [11] "tuberculose respiratória, com confirmação bacteriológica e histológica"  
#> [12] "tuberculose das vias respiratórias, sem confirmação bacteriológica ou histológica"  
#> [13] "tuberculose do sistema nervoso"  
#> [14] "tuberculose de outros órgãos"  
#> [15] "tuberculose miliar"
```

Necessitaremos corrigir a escrita, transformando todos as letras em frases, ou seja, vamos deixar apenas a primeira maiúscula e o restante das palavras em minúsculas utilizando a função `str_to_sentence()`.

Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Transformando apenas a primeira maiúscula e restante das palavras em minúsculas  
str_to_sentence(nomes_cid)
```



```
#> [1] "Cólera"
#> [2] "Febres tifóide e paratifóide"
#> [3] "Outras infecções por salmonella"
#> [4] "Shigelose"
#> [5] "Outras infecções intestinais bacterianas"
#> [6] "Outras intoxicações alimentares bacterianas, não classificadas em outra parte"
#> [7] "Amebíase"
#> [8] "Outras doenças intestinais por protozoários"
#> [9] "Infecções intestinais virais, outras e as não especificadas"
#> [10] "Diarréia e gastroenterite de origem infecciosa presumível"
#> [11] "Tuberculose respiratória, com confirmação bacteriológica e histológica"
#> [12] "Tuberculose das vias respiratórias, sem confirmação bacteriológica ou histológica"
#> [13] "Tuberculose do sistema nervoso"
#> [14] "Tuberculose de outros órgãos"
#> [15] "Tuberculose miliar"
```

Mas e se você necessitasse corrigir erros nos textos? Na rotina de vigilância em saúde, ao manipular bancos de dados com campos de preenchimento de texto aberto - como por exemplo os campos "observação" - é comum observarmos uma mesma palavra grafada de diferentes formas. Vamos aprender aqui como corrigir estes problemas de escrita.

Imagine que, analisando o campo de descrição do campo "observação" da notificação de uma doença ou agravo, você observou que alguns dos registros contêm de maneira frequente a palavra "Infecções" e ao exportar o banco de dados ela aparece escrita de diferentes maneiras:

"Infecções" (escrita corretamente), "Infeccões" (sem o cedilha), "Infeçoes" (sem o acento) e "Infeccoes" (sem acento e cedilha).

Sabemos que, neste caso, todas essas grafias se referem a uma mesma palavra.

Mas você deve estar se perguntando: "Como podemos padronizar esta palavra em nossos bancos?". Uma solução bastante simples é padronizarmos utilizar a palavra sem nenhum acento ou caractere especial (no caso, o "ç"). Isto pode ser feito utilizando a função `stri_trans_general()` do pacote `stringi`. O segundo argumento (`id = "Latin-ASCII"`) indica a transformação de caracteres latinos para o padrão ASCII, que será responsável pela padronização dos caracteres com símbolos especiais para letras simples.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Vetor com diferentes grafias
nomes <- c("infecções", "infeccões", "infecções", "infeccoes")

# Transformação das letras especiais ou com acentuação
stri_trans_general(nomes, id = "Latin-ASCII")
```

```
#> [1] "infeccoes" "infeccoes" "infeccoes" "infeccoes"
```

Veja que como resultado você obtém todas as palavras escritas da mesma maneira.

Quase sempre precisamos obter informações de campos abertos de formulários, seja porque possuem informações sobre o tratamento do paciente ou sobre quem o atendeu, ou mesmo de sinais e sintomas que não puderam ser adicionados em campos fechados. Agora você é capaz de resolver o problema com a grafia dos campos de “observação” de formulários na área da saúde. Pratique!

8.5 Identificando padrões

E se você quisesse saber quantas vezes a palavra “tuberculose” foi digitada no campo observação de um formulário? Ou mesmo, se você necessita avaliar quais os agravos em que esta palavra foi descrita?

Para responder a essas perguntas, iremos te ensinar como identificar padrões em textos utilizando o pacote `stringr` e a sua função `str_subset()`. Esta função possui dois argumentos principais:

- `string`: um string ou vetor de strings,
- `pattern`: o padrão a ser identificado.

Vamos praticar avaliando o vetor de nomes da CID ({nomes_cid}) criado anteriormente. Observe em seu *output* a visualização das linhas que contém a palavra “Tuberculose”.

Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Extraindo linhas onde a palavra "Tuberculose" aparece  
str_subset(nomes_cid, pattern = "Tuberculose")
```

```
#> [1] "Tuberculose respiratória, com confirmação bacteriológica e histológica"  
#> [2] "Tuberculose das vias respiratórias, sem confirmação bacteriológica  
ou histológica"  
#> [3] "Tuberculose do sistema nervoso"  
#> [4] "Tuberculose de outros órgãos"  
#> [5] "Tuberculose miliar"
```

Para ignorar as letras maiúsculas e minúsculas é possível utilizar a função `fixed()` dentro do argumento `pattern`. Essa função possui o argumento `ignore_case` (ignorar maiúsculas ou minúsculas). Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Extraindo linhas onde a palavra "tuberculose" aparece, independente se  
# maiúscula ou minúscula  
str_subset(nomes_cid, pattern = fixed("tuberculose", ignore_case = TRUE))
```

```
#> [1] "Tuberculose respiratória, com confirmação bacteriológica e histológica"  
#> [2] "Tuberculose das vias respiratórias, sem confirmação bacteriológica ou histológica"  
#> [3] "Tuberculose do sistema nervoso"  
#> [4] "Tuberculose de outros órgãos"  
#> [5] "Tuberculose miliar"
```

Como resultado, você verá um novo vetor contendo apenas os elementos descritos na expressão escolhida.



A função `str_detect()` procura pelo padrão em qualquer posição do texto. Existem duas funções alternativas que você pode utilizar quando quer encontrar padrões no início ou ao final do texto. Elas são respectivamente `str_start()` e `str_end()`. Neste exemplo, iremos procurar pela palavra independente se for maiúscula ou minúscula.

Acompanhe o script abaixo e replique-o em seu **RStudio**:

```
CID10 |>

# Utilizando a função `filter()` com a função `str_starts()`
filter(
  str_starts(

    # Selecionando a variável usada para encontrar palavras no início da frase
    string = DESCRICAO,

    # Definindo a palavra a ser pesquisada e ignorando se maiúscula ou minúscula
    pattern = fixed("outras", ignore_case = TRUE)
  )
) |>

# Selecionando as variáveis que queremos utilizar com a função `select()`
select(CAT, DESCRICAO)
```

```
#> # A tibble: 179 × 2
#>   CAT   DESCRIÇÃO
#>   <chr> <chr>
#> 1 A02   Outras infecções por Salmonella
#> 2 A04   Outras infecções intestinais bacterianas
#> 3 A05   Outras intoxicações alimentares bacterianas,
não classificadas em outr...
#> 4 A07   Outras doenças intestinais por protozoários
#> 5 A28   Outras doenças bacterianas zoonóticas não
classificadas em outra parte
#> 6 A41   Outras septicemias
#> 7 A48   Outras doenças bacterianas não classificadas em
outra parte
#> 8 A53   Outras formas e as não especificadas da sífilis
#> 9 A56   Outras infecções causadas por clamídias
transmitidas por via sexual
#> 10 A63  Outras doenças de transmissão predominantemente
sexual, não classifica...
#> # ... with 169 more rows
```

Para selecionar apenas quando uma expressão ocorre ao final do *string*, utilize `str_ends()` e para buscar no início utilize `str_starts()`. Acompanhe o script abaixo e replique-o em seu RStudio:

```
CID10 |>
```

```
# Utilizando a função `filter()` com a função `str_ends()`  
filter(  
  str_ends(  
  
    # Definindo a variável usada para encontrar palavras no final da frase  
    string = DESCRICA0,  
  
    # Definindo a palavra a ser pesquisada e ignorando se maiúscula ou minúscula  
    pattern = fixed("bacterianas", ignore_case =  
TRUE)  
  )  
) |>  
  
# Seleccionando as variáveis que queremos utilizar com a função `select()`  
select(CAT, DESCRICA0)
```

```
#> # A tibble: 2 × 2  
#>   CAT   DESCRICA0  
#>   <chr> <chr>  
#> 1 A04   Outras infecções intestinais bacterianas  
#> 2 Y58   Efeitos adversos de vacinas bacterianas
```

Pronto, agora você já sabe como seria refinar a busca de palavras em campos abertos de autopreenchimento.

8.6 Substituindo valores de texto

O pacote `stringr` apresenta diversas possibilidades de transformação de textos. A mais simples dela permite que uma substituição de caracteres seja feita de acordo com a posição dos caracteres no *string*.

Imagine que a frase “Tuberculose respiratória, com confirmação bacteriológica e histológica” foi escrita de maneira errônea e você necessita substituir o texto contido nela devendo trocar a palavra “com” pela palavra “sem”. Para isso utilizaremos a função `str_replace()`.

Esta função busca no texto a expressão de interesse e a substitui diretamente, sem necessitar localizar a posição do caracterer na palavra a ser substituída. Vamos praticar! Você deverá utilizar os três argumentos de `str_replace()`:

- `string`: string ou vetor de strings que será utilizado para a substituição,
- `pattern`: o padrão de texto a ser identificado para substituição,
- `replacement`: o string de texto que irá substituir o padrão.

É importante ressaltar também que esta função exige o uso do operador `<-` acompanhado da expressão que será utilizada para a substituição. Veja abaixo como ficaria a substituição da palavra “com” pelo termo “sem”, utilizando a função `str_replace()`.

Acompanhe o *script* e replique-o em seu `RStudio`:

```
# Criando um vetor de texto
agravo <- "Tuberculose respiratória, com confirmação bacteriológica e
histológica"

# Substituindo palavras
str_replace(agravo, pattern = "com", replacement = "sem")
```

```
#> [1] "Tuberculose respiratória, sem confirmação bacteriológica e histológica"
```

Note que a função `str_replace()` irá substituir apenas a primeira vez em que encontrar um padrão. Para que possamos substituir um padrão todas as vezes que ele aparecer no texto, utilize a função `str_replace_all()`. No exemplo a seguir, vamos utilizar o mesmo agravo visto no último exemplo e substituir todas as expressões contendo o acento “ó” pela vogal “o”, sem acento.

Acompanhe o *script* abaixo e replique-o em seu `RStudio`:

```
# Criando um vetor de texto
agravo <- "Tuberculose respiratória, com confirmação bacteriológica e
histológica"

# Substituindo palavras
str_replace_all(agravo, pattern = "ó", replacement = "o")
```

```
#> [1] "Tuberculose respiratoria, com confirmação bacteriologica e histologica"
```


9. Organizando os eventos de vigilância no tempo

Muitas vezes para a interpretação dos dados e construção de hipóteses em uma investigação epidemiológica, o profissional de vigilância necessita considerar uma série de acontecimentos baseados nas informações obtidas anteriormente.

Esta etapa do curso apresentará metodologias sistematizadas utilizando a linguagem **R** que permitirão com rapidez antever futuros cenários da distribuição de doenças. Por exemplo, avaliando se a epidemia está em ascensão ou declínio, se tem períodos (dias, semanas, meses ou anos) de remissão ou até mesmo recrudescimento de casos. Ou seja, você será capaz de conhecer a distribuição de uma doença, permitindo tomar decisões oportunas e em tempo hábil.

Aqui você aprenderá a organizar seus dados por dia, mês, ano e semana epidemiológica com facilidade e rapidez dentro do **R** utilizando o pacote **lubridate**.

Agora faremos uma série de exercícios em que aprenderemos métodos que facilitem conhecer e organizar a distribuição de um evento segundo suas características no tempo.

9.1 Transformando datas

O primeiro passo para se trabalhar com datas é entender o formato em que o **R** costuma interpretar datas. Como definição padrão, as datas no **R** são representadas no seguinte formato:

YYYY-MM-DD

Sendo YYYY o ano (*year*, em inglês) com quatro dígitos, MM os meses (*month*, em inglês), e DD os dias (*day*, em inglês). Cada uma dessas informações é separada por um hífen (-).

É possível criar um objeto de data de R a partir de um string de texto. Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Transformando uma string em data
data_1 <- as_date("2022-05-19")

# Visualizando a string transformada em data
data_1
```

```
#> [1] "2022-05-19"
```

Como resultado, você verá a data digitada entre aspas.

Agora vamos utilizar a função `class()` para verificar o formato (o tipo de dado) da tabela `data_1`. Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Verificando a classe do objeto `data_1`
class(data_1)
```

```
#> [1] "Date"
```

Perceba que ela está no formato **Date**. Entretanto, quando trabalhamos com diferentes bancos de dados, as datas podem se apresentar nos mais diferentes formatos. No Brasil é comum utilizarmos datas no formato **DD/MM/AAAA** (por exemplo: "19/05/2022"). Os dados exportados do Sistema de Informação sobre Mortalidade, por exemplo, têm as datas apresentadas sem separação, no formato **DDMMAAAA** (19052022), o que dificulta o manuseio destas datas no dia a dia de forma rápida.

Para que o R interprete estas datas corretamente, precisaremos definir o formato de data que deverá ser considerado. Isto é definido pelo uso de símbolos, conforme demonstrado na tabela a seguir:

| Símbolo | Descrição | Exemplos |
|---------|------------------------------------|---------------------------|
| %Y | Ano completo | 1997, 2001 |
| %y | Apenas os 2 últimos dígitos do ano | 97, 01 |
| %m | Mês (dígitos) | 01, 09, 12 |
| %b | Mês (abreviado) | Jan, Ago, Dez |
| %B | Mês (completo) | Janeiro, Agosto, Dezembro |
| %d | Dia do mês | 03, 17, 28 |

Agora vamos estudar alguns exemplos. Observe que ao utilizarmos a data no formato **DD/MM/AAAA**, com a função `as_date()`, visualizaremos uma mensagem de aviso (*warning*) e o objeto resultante será o valor **NA**.

Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Transformando uma string em data sem definir o formato  
as_date("19/05/2022")
```

```
#> Warning: All formats failed to parse. No formats found.
```

```
#> [1] NA
```

Para que o código interprete corretamente a data a ser transformada, devemos utilizar o argumento `format`, e entre aspas colocar o código correspondentes ao formato de data que queremos que a função interprete. Desta forma, para que a função compreenda que o formato de data que queremos é **"DD/MM/AAAA"**, devemos utilizar o código `%d/%m/%Y` desta forma.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Transformando uma string em data definindo o formato  
as_date("19/05/2022", format = "%d/%m/%Y")
```

```
#> [1] "2022-05-19"
```

Note que o R irá retornar a data no formato que utiliza como padrão: **AAAA-MM-DD**, ou como podemos compreender agora, %Y-%m-%d. Para resolver o problema de datas como utilizado no banco de dados SIM, **DDMMAAAA**, acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Transformando uma string do tipo das datas do SIM definindo o formato  
as_date("01031998", format = "%d%m%Y")
```

```
#> [1] "1998-03-01"
```

Estas transformações são muito importantes quando trabalhamos no R. Datas salvas como *strings* limitarão alguns cálculos (como cálculo entre datas, ou séries temporais). Assim ao final de qualquer manipulação de dados é preciso verificar se as colunas que armazenam datas são do tipo "data" (**Date**). Caso não sejam, você deve transformá-las para este formato.

9.2 Cálculo com datas

Agora que aprendemos a extrair datas e transformá-las em formatos desejados, podemos com segurança, realizar diferentes operações com as datas.

Para os próximos exemplos iremos retornar o uso do banco de dados {`NINDINET.dbf`}. Em especial, utilizaremos os 10 primeiros registros das colunas `DT_NOTIFIC`, `DT_SIN_PRI`, e `DT_NASC`, contendo a data de notificação de agravo, a data de primeiros sintomas, e a data de nascimento, respectivamente. Siga a o passo a passo abaixo:

1. importe o banco de dados {`NINDINET.dbf`},
2. selecione apenas as colunas (`DT_NOTIFIC`, `DT_SIN_PRI`, e `DT_NASC`) de interesse com a função `select()`, e
3. escolha os 10 primeiros registros com a função `slice()`.

Acompanhe o *script* abaixo e replique-o em seu `RStudio`

```
# Importando o banco de dados { `NINDINET.dbf` } para o `R`  
dt_notific <- read.dbf(file = 'Dados/NINDINET.dbf') |>  
  
# Selecionando as variáveis que queremos utilizar com a função `select()`  
select(DT_NOTIFIC, DT_SIN_PRI, DT_NASC) |>  
  
# Selecionar apenas as linhas de interesse (linhas 1 a 10)  
slice(1:10)
```

Caso deseje, visualize os valores selecionados apenas digitando o nome do objeto criado. Acompanhe o *script* abaixo e replique-o em seu `RStudio`:

```
# Visualizando o objeto {`dt_notific`}  
dt_notific
```

```
#>   DT_NOTIFIC DT_SIN_PRI   DT_NASC
#> 1  2012-04-11 2012-04-05 2012-04-04
#> 2  2010-09-17 2010-09-09 1988-04-23
#> 3  2010-10-19 2010-10-19 1971-03-25
#> 4  2008-04-14 2008-04-11 1928-05-29
#> 5  2011-06-20 2011-04-02 2002-09-18
#> 6  2008-02-12 2008-02-06 1953-08-01
#> 7  2007-12-14 2007-12-03 1975-10-20
#> 8  2011-07-06 2011-07-06 1996-08-14
#> 9  2008-04-24 2008-04-23 2000-10-28
#> 10 2011-07-06 2011-07-06 2000-03-03
```

A operação mais simples que veremos aqui é a de soma envolvendo datas. Somando um valor inteiro a uma data, iremos calcular a data correspondente após a soma deste número de dias. Esse exercício é importante para avaliar, por exemplo, a data de encerramento da maioria dos agravos de notificação compulsória a partir da data de notificação. Acompanhe o *script* abaixo que adicionamos 60 dias à data de notificação. Replique-o em seu RStudio:

```
# Somando 60 dias às datas de notificação de casos da tabela {`dt_notific`}
dt_notific$DT_NOTIFIC + 60
```

```
#> [1] "2012-06-10" "2010-11-16" "2010-12-18" "2008-06-13" "2011-08-19"
#> [6] "2008-04-12" "2008-02-12" "2011-09-04" "2008-06-23" "2011-09-04"
```

Da mesma forma, é possível subtrair 60 dias as datas de notificação da tabela. Perceba que a mesma lógica se aplica quando fazemos a subtração de datas. Acompanhe abaixo o *script* e replique-o em seu RStudio:

```
# Subtraindo 60 dias as datas de notificação
#de casos à tabela {`dt_notific`}
dt_notific$DT_NOTIFIC - 60
```

```
#> [1] "2012-02-11" "2010-07-19" "2010-08-20" "2008-02-14" "2011-04-21"  
#> [6] "2007-12-14" "2007-10-15" "2011-05-07" "2008-02-24" "2011-05-07"
```

Agora faremos a comparação entre as datas calculadas acima utilizando a função `mutate()` após selecionar apenas a data de notificação dos casos (coluna `DT_NOTIFIC`).

Acompanhe o *script* abaixo e replique-o em seu `RStudio`:

```
# Criando a tabela {`dt_notific_2`}
dt_notific_2 <- dt_notific |>

# Selecionando as variáveis que queremos utilizar com a função `select()`
select(DT_NOTIFIC) |>

# Utilizando a função `mutate()` para criar a nova coluna "prazo_encerramento"
mutate(prazo_encerramento = DT_NOTIFIC + 60)

# Visualizando a tabela {`dt_notific_2`} criada
dt_notific_2
```

```
#>   DT_NOTIFIC prazo_encerramento
#> 1 2012-04-11      2012-06-10
#> 2 2010-09-17      2010-11-16
#> 3 2010-10-19      2010-12-18
#> 4 2008-04-14      2008-06-13
#> 5 2011-06-20      2011-08-19
#> 6 2008-02-12      2008-04-12
#> 7 2007-12-14      2008-02-12
#> 8 2011-07-06      2011-09-04
#> 9 2008-04-24      2008-06-23
#> 10 2011-07-06      2011-09-04
```

Como resultado você obtém uma nova tabela {`dt_notific_2`} contendo três colunas: a primeira é uma coluna índice que marca o número da linha não sendo uma variável do banco em si, a data de notificação (`DT_NOTIFIC`) e a nova coluna que se refere a data 60 dias após a data da notificação (`prazo_encerramento`).

A vigilância em saúde busca iniciar o controle e as atividades de prevenção o mais cedo possível. Uma informação importante para análise da sensibilidade do sistema de vigilância é o monitoramento do tempo transcorrido entre o início de sintomas e a data de notificação dos casos.

Com o R, podemos calcular a diferença em dias entre duas datas de forma simples e rápida. Vamos praticar!

Primeiro, criaremos o objeto `{dif_tempo}`, ele armazenará uma tabela contendo a variável `DIFERENCA`, que por sua vez guardará os valores resultantes do cálculo de diferença entre duas datas.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Criando a tabela {`dif_tempo`}
dif_tempo <- dt_notific |>

# Selecionando as variáveis que queremos utilizar com a função `select()`
select(DT_NOTIFIC, DT_SIN_PRI) |>

# Utilizando a função `mutate()` para criar a nova coluna "DIFERENCA"
mutate(DIFERENCA = DT_NOTIFIC - DT_SIN_PRI)

# Visualizando a tabela {`dif_tempo`} criada
dif_tempo
```

```
#>   DT_NOTIFIC DT_SIN_PRI DIFERENCA
#> 1 2012-04-11 2012-04-05    6 days
#> 2 2010-09-17 2010-09-09    8 days
#> 3 2010-10-19 2010-10-19    0 days
#> 4 2008-04-14 2008-04-11    3 days
#> 5 2011-06-20 2011-04-02   79 days
#> 6 2008-02-12 2008-02-06    6 days
#> 7 2007-12-14 2007-12-03   11 days
#> 8 2011-07-06 2011-07-06    0 days
#> 9 2008-04-24 2008-04-23    1 days
#> 10 2011-07-06 2011-07-06    0 days
```


Perceba que o resultado é indicado na variável **DIFERENCA** em número de dias. Para manter apenas os valores numéricos dessa diferença podemos transformar a coluna em números inteiros utilizando a função `as.integer()`.

Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Criando a tabela {`dif_tempo_2`}
dif_tempo_2 <- dt_notific |>

# Selecionando as variáveis que queremos utilizar com a função `select()`
select(DT_NOTIFIC, DT_SIN_PRI) |>

# Utilizando a função `mutate()` para criar a nova coluna "Diferenca"
mutate(Diferenca = as.integer(DT_NOTIFIC - DT_SIN_PRI))

# Visualizando a tabela {`dif_tempo_2`} criada
dif_tempo_2
```

```
#>   DT_NOTIFIC DT_SIN_PRI Diferenca
#> 1 2012-04-11 2012-04-05         6
#> 2 2010-09-17 2010-09-09         8
#> 3 2010-10-19 2010-10-19         0
#> 4 2008-04-14 2008-04-11         3
#> 5 2011-06-20 2011-04-02        79
#> 6 2008-02-12 2008-02-06         6
#> 7 2007-12-14 2007-12-03        11
#> 8 2011-07-06 2011-07-06         0
#> 9 2008-04-24 2008-04-23         1
#> 10 2011-07-06 2011-07-06         0
```

A transformação do tempo para números inteiros é importante, pois sempre necessitamos fazer outras operações com esses valores, por exemplo, quando vamos calcular a idade de cada caso notificado no momento da data de início de sintomas.

Vamos praticar mais. Observe que primeiro calcularemos a diferença em dias entre a data de nascimento e a data de primeiros sintomas para saber a idade (dias) que nossos pacientes tinham quando estavam doentes.

Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Criando a tabela {`idade`}
idade <- dt_notific |>

# Selecionando as variáveis que queremos utilizar com a função `select()`
select(DT_NASC, DT_SIN_PRI) |>

# Utilizando a função `mutate()` para criar a nova coluna "IDADE_DIAS"
mutate(IDADE_DIAS = as.integer(DT_SIN_PRI - DT_NASC))

# Visualizando a tabela {`idade`} criada
idade
```

```
#>      DT_NASC DT_SIN_PRI IDADE_DIAS
#> 1  2012-04-04 2012-04-05         1
#> 2  1988-04-23 2010-09-09       8174
#> 3  1971-03-25 2010-10-19      14453
#> 4  1928-05-29 2008-04-11      29172
#> 5  2002-09-18 2011-04-02       3118
#> 6  1953-08-01 2008-02-06      19912
#> 7  1975-10-20 2007-12-03      11732
#> 8  1996-08-14 2011-07-06       5439
#> 9  2000-10-28 2008-04-23       2734
#> 10 2000-03-03 2011-07-06      4142
```

Perceba que utilizar idade em dias pode não ser útil para suas avaliações na vigilância, portanto precisaremos dividir estas idades pelo valor por 365,25 (considera-se este valor por levar em conta os anos bissextos), e arredondaremos para o menor valor inteiro, utilizando a função `floor()`.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Alterando a tabela {`idade`}
idade <- dt_notific |>

# Selecionando as variáveis que queremos utilizar com a função `select()`
select(DT_NASC, DT_SIN_PRI) |>

# Utilizando a função `mutate()` para criar as novas colunas de idade
mutate(
  IDADE_DIAS = as.integer(DT_SIN_PRI - DT_NASC),
  IDADE_ANOS = floor(IDADE_DIAS / 365.25)
)

# Visualizando a tabela {`idade`} modificada
idade
```

```
#>      DT_NASC DT_SIN_PRI IDADE_DIAS IDADE_ANOS
#> 1 2012-04-04 2012-04-05         1         0
#> 2 1988-04-23 2010-09-09       8174        22
#> 3 1971-03-25 2010-10-19      14453        39
#> 4 1928-05-29 2008-04-11      29172        79
#> 5 2002-09-18 2011-04-02       3118         8
#> 6 1953-08-01 2008-02-06      19912        54
#> 7 1975-10-20 2007-12-03      11732        32
#> 8 1996-08-14 2011-07-06       5439        14
#> 9 2000-10-28 2008-04-23       2734         7
#> 10 2000-03-03 2011-07-06       4142        11
```

Pronto, agora será possível conhecer a idade em anos dos pacientes notificados a partir da data de início dos sintomas.

9.3 Extrair dia, mês e ano, dia da semana

Agora que aprendemos a identificar e realizar cálculos com datas, podemos extrair informações específicas delas. Esta etapa é importante, por exemplo, quando precisamos saber qual o dia da semana ou do mês possuem o maior e ou o menor número de notificações.

Para esta etapa o pacote `lubridate` apresenta funções que permitem facilmente a extração de informações como o ano, mês, dia, e dia da semana a partir de um conjunto de datas. Veja algumas formas de fazer estas extrações:

Acompanhe os *scripts* abaixo e replique-os em seu `RStudio`:

- **Extrair ano:**

```
year(dt_notific$DT_NOTIFIC)
```

```
#> [1] 2012 2010 2010 2008 2011 2008 2007 2011 2008 2011
```

- **Extrair o mês:**

```
month(dt_notific$DT_NOTIFIC)
```

```
#> [1] 4 9 10 4 6 2 12 7 4 7
```

- **Extrair o dia:**

```
day(dt_notific$DT_NOTIFIC)
```

```
#> [1] 11 17 19 14 20 12 14 6 24 6
```

• Extrair o dia da semana:

Para indicar o dia da semana utilizamos a função `wday()`. Esta função possui mais dois argumentos de interesse:

- `label`: pode ser verdadeiro (`TRUE`) ou falso (`FALSE`): indica se o valor retornado será o nome do dia da semana ou o número correspondente a este dia.
- `abbr`: pode ser verdadeiro (`TRUE`) ou falso (`FALSE`): indica se o nome do dia da semana será abreviado ou não.

Observe abaixo como realizamos os comandos abaixo e replique-os em seu `RStudio`:

• retornar apenas o número correspondente ao dia da semana

```
wday(dt_notific$DT_NOTIFIC)
```

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

• retornar o nome do dia da semana, abreviado:

```
wday(dt_notific$DT_NOTIFIC, label = TRUE)
```

```
#> [1] qua sex ter seg seg ter sex qua qui qua  
#> Levels: dom < seg < ter < qua < qui < sex < sáb
```

• retornar o nome do dia da semana, sem abreviação

```
wday(dt_notific$DT_NOTIFIC, label = TRUE, abbr = FALSE)
```

```
#> [1] quarta-feira sexta-feira terça-feira segunda-feira segunda-feira  
#> [6] terça-feira sexta-feira quarta-feira quinta-feira quarta-feira  
#> 7 Levels: domingo < segunda-feira < terça-feira < ... < sábado
```

9.4 Identificar semana epidemiológica das datas

Para a vigilância epidemiológica, uma das informações mais relevantes a respeito de uma data é a qual semana epidemiológica (SE) determinado evento pertence. A semana epidemiológica é um consenso sobre o uso de um período padrão para agrupar os casos e óbitos ou outros eventos epidemiológicos, internacionalmente.

Com o R você poderá organizar os fatos por SE com rapidez utilizando o pacote `lubridate`. Com ele você poderá identificar a semana epidemiológica a partir da função `epiweek()`. Note que esta função considera por padrão que o início da SE é o domingo.

Agora vamos organizar as datas de notificação da tabela `{dt_notific}` criada anteriormente com os dados do `{NINDINET.dbf}` em semanas epidemiológicas.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Transformando todas as datas da variável `DT_NOTIFIC`  
# da tabela `{dt_notific}` em semana epidemiológica  
# com a função `epiweek()`  
epiweek(dt_notific$DT_NOTIFIC)
```

```
#> [1] 15 37 42 16 25 7 50 27 17 27
```

Observe que para cada data você obterá um valor numérico indicando a semana epidemiológica correspondente.

Para sintetizar, vamos criar uma tabela `dt_notific_2` contendo ano, mês, dia, número da semana, nome da semana, nome completo da semana e semana epidemiológica utilizando as notificações de casos do banco de dados `{NINDINET.dbf}`.

Para isso utilizaremos a função `mutate()`, que também já foi abordada neste módulo. Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Alterando a tabela {`dt_notific_2`}
dt_notific_2 <- dt_notific |>

  # Utilizando a função `mutate()` para criar novas colunas
  mutate(

    # Criando variável ano a partir da data de notificação
    # utilizando a função `year()`
    ano = year(DT_NOTIFIC),

    # Criando variável mes a partir da data de notificação
    # utilizando a função `month()`
    mes = month(DT_NOTIFIC),

    # Criando variável dia a partir da data de notificação
    # utilizando a função `day()`
    dia = day(DT_NOTIFIC),

    # Criando variável semana_num a partir da data de notificação
    # utilizando a função `wday()`
    semana_num = wday(DT_NOTIFIC),

    # Criando variável semana_nome a partir da data de notificação
    # utilizando a função `wday()`
    semana_nome = wday(DT_NOTIFIC, label = TRUE),

    # Criando variável semana_nome_completo a partir da data de notificação
    # utilizando a função `wday()`
    semana_nome_completo = wday(DT_NOTIFIC, label = TRUE, abbr = FALSE),

    # Criando variável semana_epidemiológica a partir da data de notificação
    # utilizando a função `epiweek()`
    semana_epidemiologica = epiweek(DT_NOTIFIC)
  )

# Visualizando a tabela {`dt_notific_2`} alterada
dt_notific_2
```

```
#>   DT_NOTIFIC DT_SIN_PRI   DT_NASC  ano mes dia semana_num semana_nome
#> 1 2012-04-11 2012-04-05 2012-04-04 2012  4 11         4         qua
#> 2 2010-09-17 2010-09-09 1988-04-23 2010  9 17         6         sex
#> 3 2010-10-19 2010-10-19 1971-03-25 2010 10 19         3         ter
#> 4 2008-04-14 2008-04-11 1928-05-29 2008  4 14         2         seg
#> 5 2011-06-20 2011-04-02 2002-09-18 2011  6 20         2         seg
#> 6 2008-02-12 2008-02-06 1953-08-01 2008  2 12         3         ter
#> 7 2007-12-14 2007-12-03 1975-10-20 2007 12 14         6         sex
#> 8 2011-07-06 2011-07-06 1996-08-14 2011  7  6         4         qua
#> 9 2008-04-24 2008-04-23 2000-10-28 2008  4 24         5         qui
#> 10 2011-07-06 2011-07-06 2000-03-03 2011  7  6         4         qua
#>   semana_nome_completo semana_epidemiologica
#> 1          quarta-feira                   15
#> 2          sexta-feira                   37
#> 3          terça-feira                   42
#> 4          segunda-feira                   16
#> 5          segunda-feira                   25
#> 6          terça-feira                    7
#> 7          sexta-feira                   50
#> 8          quarta-feira                   27
#> 9          quinta-feira                   17
#> 10         quarta-feira                   27
```

Dependendo do tipo de análise a ser realizada ou da necessidade de visualização, pode ser importante agregar dados em semanas epidemiológicas, e então indicar a data de final desta semana epidemiológica. Podemos fazer isso com um pequeno truque.

Primeiro, calculamos o número correspondendo ao dia da semana, sendo 1 referente ao domingo (primeiro dia da semana), e 7 referente ao sábado (o último dia).

Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Criando a tabela {`dt_notific_3`}
dt_notific3 <- dt_notific |>

# Selecionando as variáveis que queremos utilizar com a função `select()`
select(DT_NOTIFIC) |>

# Utilizando a função `mutate()` para criar a variável "dia_semana"
mutate(dia_semana = wday(DT_NOTIFIC))

# Visualizando o objeto criado
dt_notific3
```

```
#>   DT_NOTIFIC dia_semana
#> 1 2012-04-11         4
#> 2 2010-09-17         6
#> 3 2010-10-19         3
#> 4 2008-04-14         2
#> 5 2011-06-20         2
#> 6 2008-02-12         3
#> 7 2007-12-14         6
#> 8 2011-07-06         4
#> 9 2008-04-24         5
#> 10 2011-07-06        4
```

Em seguida, calculamos a diferença deste dia em relação ao último (sábado), subtraindo de 7 os valores encontrados. Acompanhe o *script* abaixo e replique-o em seu **RStudio**:

```
# Criando a tabela {`dt_notific_4`}
dt_notific_4 <- dt_notific |>

# Selecionando as variáveis que queremos utilizar com a função `select()`
select(DT_NOTIFIC) |>

# Utilizando a função `mutate()` para criar as novas variáveis
mutate(dia_semana = wday(DT_NOTIFIC),
       dif_dia_semana = 7 - dia_semana)

# Visualizando a tabela {`dt_notific_4`} criada
dt_notific_4
```

```
#>   DT_NOTIFIC dia_semana dif_dia_semana
#> 1  2012-04-11         4             3
#> 2  2010-09-17         6             1
#> 3  2010-10-19         3             4
#> 4  2008-04-14         2             5
#> 5  2011-06-20         2             5
#> 6  2008-02-12         3             4
#> 7  2007-12-14         6             1
#> 8  2011-07-06         4             3
#> 9  2008-04-24         5             2
#> 10 2011-07-06         4             3
```

Por último, somaremos esta diferença (`dif_dia_semana`) à data do evento (`DT_NOTIFIC`). Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Criando a tabela {`dt_notific_5`}
dt_notific_5 <- dt_notific |>
  # Selecionando as variáveis que queremos utilizar com a função `select()`
  select(DT_NOTIFIC) |>

# Utilizando a função `mutate()` para criar as novas variáveis
mutate(
  dia_semana = wday(DT_NOTIFIC),
  dif_dia_semana = 7 - dia_semana,
  DT_semana_epi = DT_NOTIFIC + dif_dia_semana
)

# Visualizando a tabela {`dt_notific_5`} criada
dt_notific_5
```

```
#>   DT_NOTIFIC dia_semana dif_dia_semana DT_semana_epi
#> 1  2012-04-11         4             3   2012-04-14
#> 2  2010-09-17         6             1   2010-09-18
#> 3  2010-10-19         3             4   2010-10-23
#> 4  2008-04-14         2             5   2008-04-19
#> 5  2011-06-20         2             5   2011-06-25
#> 6  2008-02-12         3             4   2008-02-16
#> 7  2007-12-14         6             1   2007-12-15
#> 8  2011-07-06         4             3   2011-07-09
#> 9  2008-04-24         5             2   2008-04-26
#> 10 2011-07-06         4             3   2011-07-09
```

Naturalmente, podemos realizar todos estes cálculos criando apenas uma coluna nova ao final da tabela `dt_notific_6`. Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Criando a tabela {`dt_notific_6`}
dt_notific_6 <- dt_notific |>

# Selecionando as variáveis que queremos utilizar com a função `select()`
select(DT_NOTIFIC) |>

# Utilizando a função `mutate()` para criar a nova coluna
mutate(DT_semana_epi = DT_NOTIFIC + 7 - wday(DT_NOTIFIC))

# Visualizando o objeto criado
dt_notific_6
```

```
#>   DT_NOTIFIC DT_semana_epi
#> 1 2012-04-11 2012-04-14
#> 2 2010-09-17 2010-09-18
#> 3 2010-10-19 2010-10-23
#> 4 2008-04-14 2008-04-19
#> 5 2011-06-20 2011-06-25
#> 6 2008-02-12 2008-02-16
#> 7 2007-12-14 2007-12-15
#> 8 2011-07-06 2011-07-09
#> 9 2008-04-24 2008-04-26
#> 10 2011-07-06 2011-07-09
```

9.5 Sequência de datas

Em certas situações, pode ser necessária a criação de uma sequência de dados em intervalos regulares. Uma situação mais simples é a situação em que queremos criar uma sequência de dias ao longo de uma semana a partir de uma data inicial.

Como havíamos visto anteriormente, o R permite que somemos uma data a um valor numérico inteiro. Desta forma, podemos criar um vetor com valores no total de 7 datas (0 a 7), e somar a uma data que tivermos interesse. Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Criando uma sequência de datas, criando dias de uma semana  
as.Date("2022-01-01") + 0:7
```

```
#> [1] "2022-01-01" "2022-01-02" "2022-01-03" "2022-01-04" "2022-01-05"  
#> [6] "2022-01-06" "2022-01-07" "2022-01-08"
```

Para situações mais complexas, também podemos utilizar a função `seq.Date()` para sequências personalizadas. Considere que você precise criar uma sequência de datas entre o primeiro dia de dois meses consecutivos: 1º de janeiro de 2022 e 1º de fevereiro de 2022. A função `seq.Date()` possui três argumentos:

- **from**: data de início da sequência,
- **to**: data final da sequência,
- **by**: intervalo de tempo entre as duas datas.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
# Criando uma sequência de datas  
seq.Date(from = as.Date("2022-01-01"), to = as.Date("2022-02-01"), by = "day")
```

```
#> [1] "2022-01-01" "2022-01-02" "2022-01-03" "2022-01-04" "2022-01-05"  
#> [6] "2022-01-06" "2022-01-07" "2022-01-08" "2022-01-09" "2022-01-10"  
#> [11] "2022-01-11" "2022-01-12" "2022-01-13" "2022-01-14" "2022-01-15"  
#> [16] "2022-01-16" "2022-01-17" "2022-01-18" "2022-01-19" "2022-01-20"  
#> [21] "2022-01-21" "2022-01-22" "2022-01-23" "2022-01-24" "2022-01-25"  
#> [26] "2022-01-26" "2022-01-27" "2022-01-28" "2022-01-29" "2022-01-30"  
#> [31] "2022-01-31" "2022-02-01"
```

A facilidade desta função está na permissão de especificar intervalos diferentes no argumento `by`, contendo os argumentos:

- `day`: intervalo em dias
- `week`: intervalo em semanas
- `month`: intervalo em mês
- `quarter`: intervalo a cada três meses
- `year`: intervalo em anos

Pratique as transformações de datas no seu dia a dia!

10. Exportação dos dados

Após manipularmos dados no software R, podemos criar tabelas de interesse e exportá-las no formato necessário. Veremos neste tópico os principais formatos para exportar seus dados.

10.1 Exportação para a extensão CSV

O pacote `readr` oferece algumas opções de exportação de dados diretamente do ambiente de trabalho. Suas funções são análogas às funções nativas de exportação.

Neste tópico iremos apresentar a função `write_csv()`. Esta possui dois argumentos principais:

- **x**: recebe o objeto (`data.frame`, `tibble`, etc) que será exportado do R e;
- **file**: o caminho onde será salvo e o nome do arquivo contendo a extensão.

Veja um exemplo de exportação de uma das bases que montamos neste módulo para a pasta do diretório “Meus documentos” do seu computador.



Perceba como você escreveu a posição das barras! A escrita do caminho do arquivo é fundamental para a exportação: `"C:/Usuários/PC/Meusdocumentos/base_menor_curso.csv"`

Caso esteja utilizando a estrutura de projetos (.Rproj) não será necessário todo esse caminho, pois ele salvará no diretório padrão do projeto de forma automática.

Acompanhe o script abaixo e replique-o em seu RStudio:

```
# Salvando a tabela {base_menor} em seu computador na pasta "Meus documentos"
write_csv(x = base_menor,
          file = "C:/Usuários/PC/Meus documentos/base_menor_curso.csv")
```

Agora abra o diretório que escolheu para salvar o arquivo e veja que ele estará disponível com o nome "base_menor_curso" do tipo `.csv`.



As funções `write_csv()` e `write_csv2()` se diferem principalmente pela exportação com separador vírgula ou ponto e vírgula, respectivamente.

10.2 Exportação para a extensão XLSX

Uma opção para exportação no formato Microsoft Excel é um pacote auxiliar chamado `writexl`.

Esse pacote é muito parecido com o `readxl`, apresentado no início desse módulo, e sua principal função é `write_xlsx()`, que grava um objeto `data.frame` e `tibble` no formato `.xlsx`. Seus argumentos são parecidos com os citados anteriormente, veja:

- `x`: recebe o objeto (`data.frame`, `tibble`, etc) que será exportado do R e;
- `path`: o caminho onde será salvo e o nome do arquivo contendo a extensão `xlsx`.

Acompanhe o *script* abaixo e replique-o em seu RStudio:

```
install.packages("writexl");library(writexl)
# Salvando a tabela {base_menor} em seu computador na pasta "Meus documentos"
write_xlsx(x = base_menor,
           path = "C:/Usuários/PC/Meusdocumentos/base_menor_curso.xlsx")
```

Agora abra o diretório que escolheu para salvar o arquivo e veja que ele estará disponível com o nome “base_menor_curso” do tipo `.xlsx`.



Para mais informações sobre o tipo de arquivo disponível para exportação de dados acesse os links abaixo para consulta de pacotes que oferecem muitos recursos:

- **rio:** esse pacote muito útil que serve de ponte para outros que fazem importações e exportações. Para saber mais clique em <https://github.com/leeper/rio>;
- **haven:** importação e exportação de arquivos dos softwares SAS, STATA e SPSS. Para saber mais clique em <https://haven.tidyverse.org/>
- **Guia de importação e exportação do R:** esse guia mantido pela equipe de desenvolvimento do R amplia os tópicos de importação e exportação, apresentando vários tópicos de forma detalhada. Clique em <https://cran.r-project.org/doc/manuals/r-release/R-data.html>



Próximo módulo

Pronto, chegamos ao final do nosso módulo! Agora você é capaz de manipular seu banco de dados e transformá-lo com o apoio do R. Acesse os demais módulos deste curso para colocar em prática as análises de dados rotineiras para vigilância em saúde.

Até o próximo módulo!

