

# Curso POF: Aula 1 - Introdução ao R

Pedro Rubin

2022-05-04

## Introdução

Essa revisão de R tem o objetivo de apresentar o básico necessário para o curso em R. Em linhas gerais, é preciso saber carregar pacotes, importar arquivos csv, os verbos do dplyr (arrange, select, mutate, filter, group\_by, summarise), como juntar dois dataframes (join) e outras funções úteis que vão surgindo pelo caminho.

## Carregar pacotes e importar arquivos csv

```
library(tidyverse)
pnadc <- read_csv("./aula1/intro_r/pnadc_final.csv")
```

Para dar um panorama dos nossos dados, vamos usar a função glimpse. Vemos que, além do nome das UFs, temos diversos dados, todos separados em total (final\_t), homens (final\_h) e mulheres (final\_m).

```
glimpse(pnadc)
```

```
## Rows: 27
## Columns: 16
## $ uf_nome    <chr> "Acre", "Alagoas", "Amapa", "Amazonas", "Bahia", "Ceara", "D~
## $ pop_14_t    <dbl> 643, 2592, 612, 2960, 11817, 7224, 2452, 3177, 5563, 5287, 2~
## $ pop_14_h    <dbl> 320, 1234, 295, 1481, 5744, 3454, 1167, 1542, 2716, 2575, 13~
## $ pop_14_m    <dbl> 324, 1358, 317, 1480, 6073, 3770, 1285, 1635, 2847, 2713, 13~
## $ ft_t        <dbl> 357, 1202, 383, 1844, 7073, 4122, 1677, 2154, 3751, 2591, 18~
## $ ft_h        <dbl> 210, 711, 221, 1089, 4001, 2337, 886, 1203, 2090, 1502, 1065~
## $ ft_m        <dbl> 147, 492, 162, 755, 3072, 1785, 791, 951, 1661, 1089, 758, 6~
## $ ocup_t      <dbl> 292, 1008, 305, 1548, 5765, 3646, 1438, 1888, 3344, 2163, 16~
## $ ocup_h      <dbl> 179, 617, 182, 959, 3375, 2100, 779, 1082, 1906, 1282, 986, ~
## $ ocup_m      <dbl> 113, 391, 123, 590, 2391, 1546, 659, 807, 1439, 880, 670, 54~
## $ desocup_t   <dbl> 65, 194, 78, 296, 1307, 476, 238, 266, 407, 429, 167, 137, 1~
## $ desocup_h   <dbl> 31, 94, 39, 131, 626, 238, 107, 121, 184, 220, 79, 57, 573, ~
## $ desocup_m   <dbl> 34, 100, 39, 165, 681, 238, 132, 144, 222, 209, 88, 80, 684, ~
## $ foraft_t     <dbl> 286, 1390, 230, 1116, 4745, 3102, 775, 1023, 1813, 2696, 850~
## $ foraft_h     <dbl> 109, 523, 75, 391, 1744, 1116, 281, 339, 626, 1073, 269, 235~
## $ foraft_m     <dbl> 177, 866, 155, 724, 3001, 1986, 494, 684, 1186, 1623, 581, 4~
```

```
dim(pnadc)
```

```
## [1] 27 16
```

## Parte 1 - básico de manipulação de dados

### arrange

A função *arrange* serve para reordenar o dataframe de acordo com alguma tabela. Por padrão, ela ordena do menor para o maior. Se quisermos do De início, vejam que a tabela está ordenada alfabeticamente de acordo com o nome da UF. Assim, as 5 primeiras linhas são Acre, Alagoas, Amapá, Amazonas e Bahia

```
pnadc
```

```
## # A tibble: 27 x 16
##   uf_nome      pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t ocup_h ocup_m
##   <chr>         <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Acre           643     320     324  357  210  147   292   179   113
## 2 Alagoas       2592    1234    1358 1202  711  492  1008   617   391
## 3 Amapa         612     295     317  383  221  162   305   182   123
## 4 Amazonas     2960    1481    1480 1844 1089  755  1548   959   590
## 5 Bahia       11817    5744    6073 7073 4001 3072  5765  3375  2391
## ...
```

Se quisermos ordenar pela UF de menor população ocupada, basta fazer

```
pnadc %>% arrange(ocup_t)
```

```
## # A tibble: 27 x 16
##   uf_nome      pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t ocup_h ocup_m
##   <chr>         <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Roraima        392     196     196  257  149  109   218   130    88
## 2 Acre           643     320     324  357  210  147   292   179   113
## 3 Amapa         612     295     317  383  221  162   305   182   123
## 4 Tocantins     1192     601     590  720  429  291   631   382   249
## 5 Rondonia     1370     686     684  871  532  339   792   496   296
## ...
```

Se, ao contrário, quisermos a de maior população ocupada, é preciso incluir o *desc*

```
pnadc %>% arrange(desc(ocup_t))
```

```
## # A tibble: 27 x 16
##   uf_nome      pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t ocup_h ocup_m
##   <chr>         <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Sao Paulo    37476   18029   19447 25832 14080 11752 22322 12384  9938
## 2 Minas Gera~ 17188    8385    8803 11183  6257  4926  9926  5684  4242
## 3 Rio de Jan~ 14368    6755    7613  8992  4959  4033  7604  4316  3288
## 4 Bahia       11817    5744    6073 7073  4001 3072  5765  3375  2391
## 5 Rio Grande~  9381    4517    4864  6164  3361  2803  5668  3148  2520
## ...
```

Vejam que, nos dois casos, a ordem das linhas mudou. As colunas, por outro lado, mantiveram a mesma ordem. Ademais, não há mudança no número de linhas e colunas.

## select

A função *select* retorna um dataframe apenas com as colunas selecionadas pelo usuário. Isso pode ocorrer pelo nome ou pela posição da coluna. Para saber o nome das colunas, basta usar a função *colnames*. No exemplo, vamos selecionar a coluna *uf\_nome* (que está na posição 1) das duas formas.

```
pnadc %>% colnames()
```

```
## [1] "uf_nome" "pop_14_t" "pop_14_h" "pop_14_m" "ft_t" "ft_h"
## [7] "ft_m" "ocup_t" "ocup_h" "ocup_m" "desocup_t" "desocup_h"
## [13] "desocup_m" "foraft_t" "foraft_h" "foraft_m"
```

```
pnadc %>% select(uf_nome)
```

```
## # A tibble: 27 x 1
##   uf_nome
##   <chr>
## 1 Acre
## 2 Alagoas
## 3 Amapa
## 4 Amazonas
## 5 Bahia
## ...
```

```
pnadc %>% select(1)
```

```
## # A tibble: 27 x 1
##   uf_nome
##   <chr>
## 1 Acre
## 2 Alagoas
## 3 Amapa
## 4 Amazonas
## 5 Bahia
## ...
```

Vejam que o resultado é o mesmo. Contudo, a opção de fazer pelo nome das colunas é mais segura, porque garante que estamos pegando a coluna que queremos.

Podemos selecionar várias colunas de uma só vez

```
pnadc %>% select(uf_nome, pop_14_t, pop_14_h, pop_14_m)
```

```
## # A tibble: 27 x 4
##   uf_nome      pop_14_t pop_14_h pop_14_m
##   <chr>      <dbl>    <dbl>    <dbl>
## 1 Acre          643      320      324
## 2 Alagoas     2592     1234     1358
```

```
## 3 Amapa          612      295      317
## 4 Amazonas       2960     1481     1480
## 5 Bahia          11817    5744    6073
...
```

A função *select* também pode ser usada para reordenar as colunas. Por exemplo, se quisermos que a primeira coluna seja a força de trabalho (e o resto na mesma ordem), basta fazer

```
pnadc %>% select(ft_t, everything())
```

```
## # A tibble: 27 x 16
##   ft_t uf_nome      pop_14_t pop_14_h pop_14_m ft_h ft_m ocup_t ocup_h ocup_m
##   <dbl> <chr>      <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  357 Acre          643     320     324  210  147   292   179   113
## 2 1202 Alagoas      2592    1234    1358   711   492  1008   617   391
## 3  383 Amapa          612     295     317   221   162   305   182   123
## 4 1844 Amazonas     2960    1481    1480  1089   755  1548   959   590
## 5 7073 Bahia        11817    5744    6073  4001  3072  5765  3375  2391
...
```

Sem o *everything()*, teríamos apenas a coluna *ft\_t*

## mutate

A função *mutate* cria novas colunas. Por exemplo, temos a força de trabalho e a população ocupada. Podemos, assim, calcular a taxa de ocupação. Para isso:

```
pnadc %>%
  mutate(tx_ocup = ocup_t/ft_t)
```

```
## # A tibble: 27 x 17
##   uf_nome      pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t ocup_h ocup_m
##   <chr>      <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Acre          643     320     324  357  210  147   292   179   113
## 2 Alagoas      2592    1234    1358 1202   711   492  1008   617   391
## 3 Amapa          612     295     317  383   221   162   305   182   123
## 4 Amazonas     2960    1481    1480 1844  1089   755  1548   959   590
## 5 Bahia        11817    5744    6073 7073  4001  3072  5765  3375  2391
...
```

Onde está a coluna que criamos? Por definição, o R coloca novas colunas no fim do df. Para vê-la antes, podemos usar *select* e *everything*

```
pnadc %>%
  mutate(tx_ocup = ocup_t/ft_t) %>%
  select(uf_nome, tx_ocup, everything())
```

```
## # A tibble: 27 x 17
##   uf_nome      tx_ocup pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t ocup_h
##   <chr>      <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
```

```
## 1 Acre      0.818      643      320      324      357      210      147      292      179
## 2 Alagoas   0.839     2592     1234     1358     1202     711     492     1008     617
## 3 Amapa     0.796      612      295      317      383      221     162      305     182
## 4 Amazonas  0.839     2960     1481     1480     1844     1089     755     1548     959
## 5 Bahia     0.815    11817     5744     6073     7073     4001    3072     5765    3375
...
```

## filter

A função *filter* retorna um df apenas com as linhas que atendam a uma condição específica. Por exemplo, se quisermos apenas as UFs com força de trabalho acima de 1 milhão de pessoas, fazemos (lembrando que os dados estão em mil pessoas)

```
pnadc %>%
  filter(ft_t > 1000)
```

```
## # A tibble: 22 x 16
##   uf_nome      pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t ocup_h ocup_m
##   <chr>      <dbl>    <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Alagoas      2592      1234      1358  1202   711   492   1008    617    391
## 2 Amazonas     2960      1481      1480  1844  1089   755   1548    959    590
## 3 Bahia     11817      5744      6073  7073  4001  3072   5765   3375   2391
## 4 Ceara       7224      3454      3770  4122  2337  1785   3646   2100   1546
## 5 Distrito F~ 2452      1167      1285  1677   886   791   1438    779    659
...
```

Podemos também fazer condições com mais de uma coluna. Por exemplo, pegar apenas as UFs na qual a população desocupada feminina é inferior à masculina

```
pnadc %>%
  filter(desocup_m < desocup_h)
```

```
## # A tibble: 5 x 16
##   uf_nome      pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t ocup_h ocup_m
##   <chr>      <dbl>    <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Maranhao     5287      2575      2713  2591  1502  1089   2163   1282    880
## 2 Paraiba      3140      1507      1633  1684   994   689   1494    897    597
## 3 Piaui        2560      1256      1305  1452   854   598   1264    739    525
## 4 Rio Grande ~ 2778      1339      1438  1522   880   642   1308    761    548
## 5 Tocantins    1192       601       590   720   429   291    631    382    249
...
```

Por fim, podemos colocar mais de uma condição e combiná-las com união (ou) ou interseção (e) Como exemplo, podemos combinar as duas condições acima. Pegar apenas as UFs com força de trabalho acima de 1 milhão de pessoas e população desocupada feminina inferior à masculina

```
pnadc %>%
  filter(ft_t > 1000,
         desocup_m < desocup_h)
```

```
## # A tibble: 4 x 16
##   uf_nome      pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t ocup_h ocup_m
##   <chr>      <dbl>    <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Maranhao      5287      2575      2713 2591 1502 1089 2163 1282 880
## 2 Paraiba       3140      1507      1633 1684 994 689 1494 897 597
## 3 Piaui         2560      1256      1305 1452 854 598 1264 739 525
## 4 Rio Grande ~ 2778      1339      1438 1522 880 642 1308 761 548
## # ... with 6 more variables: desocup_t <dbl>, desocup_h <dbl>, desocup_m <dbl>,
## ...
```

Agora, pegar as UFs com força de trabalho acima de 1 milhão de pessoas ou população desocupada feminina inferior à masculina

```
pnadc %>%
  filter(ft_t > 1000 |
         desocup_m < desocup_h)
```

```
## # A tibble: 23 x 16
##   uf_nome      pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t ocup_h ocup_m
##   <chr>      <dbl>    <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Alagoas      2592      1234      1358 1202 711 492 1008 617 391
## 2 Amazonas     2960      1481      1480 1844 1089 755 1548 959 590
## 3 Bahia       11817      5744      6073 7073 4001 3072 5765 3375 2391
## 4 Ceara        7224      3454      3770 4122 2337 1785 3646 2100 1546
## 5 Distrito F~ 2452      1167      1285 1677 886 791 1438 779 659
## ...
```

## summarise

A função *summarise* serve para resumir as informações disponíveis no df. Sem outras alterações (que serão vistas mais à frente), a função *summarise* retorna um df com apenas 1 linha e quantas colunas forem criadas.

Por exemplo, se quisermos obter a soma da população ocupada de todas as UF, rodamos

```
pnadc %>%
  summarise(ocup = sum(ocup_t))
```

```
## # A tibble: 1 x 1
##   ocup
##   <dbl>
## 1 92621
```

Da mesma forma, podemos querer a média e a mediana taxa de ocupação feminina do país

```
pnadc %>%
  mutate(tx_ocup_m = ocup_m/ft_m) %>%
  summarise(media_tx_ocup_m = mean(tx_ocup_m),
            mediana_tx_ocup_m = median(tx_ocup_m))
```

```
## # A tibble: 1 x 2
##   media_tx_ocup_m mediana_tx_ocup_m
##   <dbl>          <dbl>
## 1 0.840          0.849
```

## join

A função *join* serve para juntar dois dataframes a partir de uma chave. Uma chave é uma coluna que existe nos dois dataframes e permite relacionar ambos. No caso, vamos juntar nosso df `pnadc` com o df `codigo_uf`

```
codigo_uf <- read_csv2("./aula1/intro_r/cod_final.csv")
codigo_uf %>% glimpse()
```

```
## Rows: 27
## Columns: 3
## $ uf_codigo <dbl> 11, 12, 13, 14, 15, 16, 17, 21, 22, 23, 24, 25, 26, 27, 28, ~
## $ uf_nome    <chr> "Rondonia", "Acre", "Amazonas", "Roraima", "Para", "Amapa", ~
## $ regioao    <chr> "N", "N", "N", "N", "N", "N", "N", "N", "NE", "NE", "NE", "NE", "NE", "~
```

O df `codigo_uf` tem 3 colunas: código da uf, nome da uf e região. A chave, então, é o nome da uf, que também está no df `pnadc`. Existem diversas modalidades de *join*: *full*, *inner*, *left*, *right*, *anti* e *semi*. Recomendo checar esse site para uma explicação de cada. Aqui, como todas as UFs estão em ambos os df, não faz diferença entre *full*, *inner*, *left* ou *right*. Contudo, sempre temos que ter certeza do que queremos e escolher o *join* adequado.

```
pnadc_join <- pnadc %>%
  full_join(codigo_uf, by = "uf_nome")
pnadc_join %>% dim()
```

```
## [1] 27 18
```

```
pnadc_join %>% colnames
```

```
## [1] "uf_nome" "pop_14_t" "pop_14_h" "pop_14_m" "ft_t" "ft_h"
## [7] "ft_m" "ocup_t" "ocup_h" "ocup_m" "desocup_t" "desocup_h"
## [13] "desocup_m" "foraft_t" "foraft_h" "foraft_m" "uf_codigo" "regiao"
```

Vejam que não mudou o número de linhas, mas o df `pnadc_join` tem 18 colunas: é 1 de chave (`uf_nome`), as outras 15 do `pnadc` e as outras 2 do `codigo_uf`. Vamos só reorganizar as colunas para ter o código e a região no início.

```
pnadc_join <- pnadc_join %>%
  select(uf_codigo, uf_nome, regioao, everything())
pnadc_join
```

```
## # A tibble: 27 x 18
##   uf_codigo uf_nome regioao pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t
##   <dbl> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1      12 Acre    N      643    320    324   357   210   147   292
## 2      27 Alagoas NE     2592   1234   1358  1202   711   492  1008
## 3      16 Amapa    N      612    295    317   383   221   162   305
## 4      13 Amazonas N     2960   1481   1480  1844  1089   755  1548
## 5      29 Bahia    NE    11817   5744   6073  7073  4001  3072  5765
## ...
```

## group\_by

Até agora, nós só conseguíamos fazer cálculos por UF (cada linha) ou Brasil (df completo). No entanto, nosso novo df `pnadc_join` tem a variável `região`, que permite cálculos regionais. Para fazer isso, temos duas opções. A primeira é criar 5 dfs, um para cada região. Essa abordagem faz com que cada cálculo seja repetido 5 vezes no código, tornando-o mais difícil de ler e mais propenso a erros. A segunda opção é criar um df agrupado (grouped), através da função `group_by`. No caso, vamos agrupar por região.

```
pnadc_regiao <- pnadc_join %>%
  group_by(regiao)
pnadc_regiao
```

```
## # A tibble: 27 x 18
## # Groups:   regiao [5]
##   uf_codigo uf_nome  regiao pop_14_t pop_14_h pop_14_m ft_t ft_h ft_m ocup_t
##   <dbl> <chr>    <chr>   <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1      12  Acre      N       643     320     324   357   210   147   292
## 2      27 Alagoas  NE     2592    1234    1358  1202   711   492  1008
## 3      16  Amapa     N       612     295     317   383   221   162   305
## 4      13 Amazonas N     2960    1481    1480  1844  1089   755  1548
## 5      29  Bahia     NE    11817    5744    6073  7073  4001  3072  5765
## ...
```

Visualmente, a diferença entre `pnadc_join` e `pnadc_regiao` é que no segundo aparece “Groups: regiao [5]” quando printamos. Podemos ver se um df está agrupado ou não com a função `is_grouped_df`.

```
pnadc_regiao %>% is_grouped_df()
```

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

```
pnadc_join %>% is_grouped_df()
```

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

Importante: para desfazer o `group_by`, precisamos usar a função `ungroup`

```
pnadc_regiao <- pnadc_regiao %>%
  ungroup()
pnadc_regiao %>% is_grouped_df()
```

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

Vejam que agora a função `is_grouped_df` retorna FALSE.

## group\_by + mutate

Ao usar o `group_by` com `mutate`, criamos uma nova coluna. Essa coluna vai ter o mesmo valor para os elementos de um mesmo grupo. Por exemplo, como vamos agrupar por região, todas as UF do Sul terão o mesmo valor, todas as UF do Norte terão o mesmo valor e assim por diante



```
pnadc_regiao %>%
  mutate(ocup_sem_group_by = sum(ocup_t)) %>%
  group_by(regiao) %>%
  mutate(ocup_com_group_by = sum(ocup_t)) %>%
  select(uf_codigo, uf_nome, regiao, ocup_sem_group_by, ocup_com_group_by, everything())
```

```
## # A tibble: 27 x 20
## # Groups:   regiao [5]
##   uf_codigo uf_nome  regiao ocup_sem_group_~ ocup_com_group_~ pop_14_t pop_14_h
##   <dbl> <chr>    <chr>      <dbl>          <dbl>      <dbl>    <dbl>
## 1      12 Acre      N          92621          7178        643      320
## 2      27 Alagoas NE          92621         21119       2592     1234
## 3      16 Amapa     N          92621          7178        612      295
## 4      13 Amazonas N          92621          7178       2960     1481
## 5      29 Bahia     NE          92621         21119      11817     5744
## ...
```

Vejam que a coluna `ocup_sem_group_by` tem o mesmo valor para todas as UFs. Já a coluna `ocup_com_group_by` tem um valor para Acre, Amapá e Amazonas (a população ocupada no Norte) e outro para Alagoas e Bahia (a população ocupada no Nordeste). O *mutate* não retira nenhuma coluna e nenhuma linha do df

### group\_by + summarise

Ao usar o *group\_by* com *summarise*, retornamos um novo df. Mas agora, o número de linhas é o número de categorias do nosso grupo (no caso, 5 regiões) e o número de colunas é o número de grupos (no caso, 1 - as regiões) + o número de colunas que criamos.

```
pnadc_regiao %>%
  group_by(regiao) %>%
  summarise(ocup_com_group_by = sum(ocup_t))
```

```
## # A tibble: 5 x 2
##   regiao ocup_com_group_by
##   <chr>      <dbl>
## 1 CO          7733
## 2 N           7178
## 3 NE         21119
## 4 S         14851
## 5 SE         41740
```

## Parte 1.5 (Ponte) - tópicos em boas práticas

Agora que revisamos o básico de manipulação de dados, acho importante tocar em alguns pontos do uso do R. Ao contrário da PNADC, que possui um arquivo de microdados para cada realização da pesquisa, a POF possui diversos - o que torna necessária uma discussão sobre a organização das pastas. Adicionalmente, diversos usos da POF (inclusive rendimentos e despesas) precisam da combinação desses diferentes arquivos de microdados. Então, é muito provável que o processo precise ser quebrado em diversas etapas, o que requer também uma maior organização dos scripts e dos dados.

Em linhas gerais, queremos que nosso processos em R sejam reproduzíveis, auto-contidos e transportáveis (no espaço e no tempo)

## Tudo de importante deve ser obtido por um código que está salvo

Todos os arquivos e figuras importantes devem ser salvos em arquivos separados, por meio de comandos explícitos no código salvo. Em outras palavras, não salvar como parte do workspace, nem salvar com o mouse, nem por um comando no console. Isso é extremamente importante para reprodutibilidade, pois garante que o processo que gerou e salvou aquela tabela ou gráfico pode ser utilizado novamente (se o arquivo final for deletado sem querer, ou se mudanças forem necessárias).

Para tal, é importante sempre começar o R do zero (para garantir que tudo o que é preciso para fazer o código funcionar está sendo feito no próprio código). Portanto, não salvar nem carregar arquivos *.Rdata*.

No RStudio, ir em *Tools > Global Options*: desmarcar *Restore .RData into workspace at startup* e colocar *Never* em *Save workspace to .RData on exit*.

## Reiniciar o R frequentemente / Por que é ruim usar `rm(list = ls())`?

A continuação do ponto anterior - sempre começar o R do zero - é que é uma boa prática reiniciar periodicamente o R ao longo de um processo de programação. Para isso, é muito comum ver o seguinte comando no início do código: `rm(list = ls())`. Isso é um problema porque, basicamente, esse comando não reinicia o R de fato. Ele apenas deleta objetos criados pelo usuário. Todos os pacotes já carregados continuam ativos, assim como diversas outras alterações. Não há nada de errado com `rm(list = ls())` - a questão é achar que isso garante a limpeza completa do R.

Como exemplo, suponho que você está trabalhando com um script, no qual carrega o tidyverse `library(tidyverse)`. No meio do caminho, você decide criar um novo script para realizar cálculos auxiliares. Para “reiniciar” o R, você usa `rm(list = ls())`. Como vimos anteriormente, isso não coloca o R do zero, então as funções do tidyverse continuam ativas mesmo sem um chamado `library(tidyverse)` nesse segundo script - por exemplo, é possível criar uma coluna com a função `mutate`.

Se por acaso você decide revisitar esse segundo script em outro momento (ou envia a uma pessoa), o script não vai rodar! Porque a função `mutate` não existe sem que o tidyverse (mais especificamente, o dplyr) esteja carregado! Isso pode criar um enorme problema, dependendo do tamanho do script e dos pacotes utilizados. Tudo isso poderia ter sido evitado com o reinício correto do R - o problema no segundo script seria encontrado (e possivelmente resolvido) na hora.

**Para reiniciar de verdade, basta ir em *Session > Restart R* ou usar o atalho **Control+Shift+F10** (Windows e Linux) ou **Command+Shift+F10** (Mac OS).**

Em uma análise que leva tempo para carregar os dados e realizar os cálculos (como é o caso da POF), a perspectiva de reiniciar constantemente o R é um pouco assustadora. Nesses casos, a melhor alternativa é dividir o processo em várias etapas (em diferentes scripts).

## Projetos e os problemas de `setwd()`

Há, no entanto, uma limitação quando reiniciamos o R (a partir de agora, assumo que é reiniciar de fato) - isso não muda o diretório ao qual o R está se referindo. Isso levanta as questões de como definir o diretório do R e de como organizar os arquivos necessários para realizar uma análise.

Começando pela segunda, a abordagem favorecida aqui é a de criar um projeto (letra minúscula). Um projeto consiste em reunir, em uma mesma pasta, todos os arquivos necessários para o trabalho no R. Isso inclui os dados crus, os scripts e quaisquer resultados gerados (tabelas, gráficos, imagens etc.). Subpastas podem (e devem) ser utilizadas para melhorar a organização. Por exemplo, podemos ter uma pasta só para os dados iniciais, outra só para os scripts de R e por aí vai. A configuração exata depende do gosto pessoal e das necessidades do trabalho. Fazendo isso, garantimos que nossa análise está auto-contida e pode ser transferida sem grandes questões.

Exceto a enorme questão de como definir o diretório de trabalho do R. Basicamente, o diretório deve ser especificado como a pasta-mãe que contém todos (e apenas) os arquivos que utilizamos no trabalho em questão. Para dizer ao R qual é essa pasta, temos ao menos duas opções: `setwd()` e usar *RProject*. Pelo título, dá para ver que eu privilegio a segunda.

## **setwd()**

Muita gente usa *setwd* no início do script para definir o diretório de trabalho. Basicamente, essa abordagem consiste em explicitar o caminho da pasta-mãe em seu computador. Por exemplo, no meu computador, o diretório desse documento é: `setwd("C:\\Users\\prubi\\Desktop\\curso_pof")` Isso é conhecido como um caminho absoluto até a pasta.

O principal problema de definir o diretório a partir do *setwd* é que a chance de isso funcionar em qualquer outro computador é essencialmente 0 - a não ser que alguém tenha o exato mesmo caminho que eu. Então, nossa análise deixa de estar auto-contida, pois faz referência a algo fora da pasta-mãe. Ela também não é reprodutível: qualquer pessoa que tente rodar esse script não vai conseguir, a não ser que mude o diretório em todos os scripts relevantes. Isso afeta a própria pessoa que criou do script: se mudar de computador ou reorganizar os arquivos, nada mais vai rodar direito.

Então, por mais que o *setwd* cumpra seu papel, ele cria as condições para surgirem problemas no futuro.

## **Utilizando um RProject**

O RStudio tem uma implementação da ideia de projeto, o RProject. Uma vez que temos um projeto estruturado (lembrando, uma pasta mãe que contém todos e apenas os arquivos necessários para a análise), podemos associar a pasta mãe a um RProject. Com isso feito, o RStudio cria um arquivo `.Rproj` na pasta mãe. Se abrirmos o R por meio desse arquivo `.Rproj`, o diretório estará automaticamente configurado na pasta mãe, e não precisamos mais usar `setwd()`. Cumprindo todas essas etapas - estruturar nossos dados em um projeto e criar um RProject na pasta mãe (além dos outros passos anteriores) - não precisamos de qualquer referência a algo de fora do projeto, tornando-o auto-contido. Da mesma forma, como não é mais necessário estabelecer o diretório por um caminho absoluto que é virtualmente único para cada computador, diversas pessoas podem rodar o mesmo script sem problemas, tornando-o transportável e reprodutível.

Recapitulando, ao abrir o R por meio de qualquer RProject, automaticamente o diretório é definido com a pasta na qual está o arquivo `.Rproj`. Se organizarmos direito, esta é a pasta mãe de nossa análise. A partir daí, qualquer referência a arquivos ocorrerá por caminhos relativos (por exemplo `"/dados/tabela1.csv"`). Então, para compartilhar nossas análises, basta compartilhar o projeto (isto é, a pasta mãe e todas as subpastas).

Para criar um RProject, basta ir em *File > New Project...* O RProject pode ser criar uma nova pasta ou ser associado a uma pasta já existente. Para abrir o RProject, basta clicar no arquivo `.Rproj` ou abrir o R e ir em *File > Open Project* ou clicar no ícone do RProject que está no canto superior direito do RStudio.

## **Parte 2 - Carregando a POF**

Como vimos anteriormente, a POF é composta de diversos registros, cada um com seu arquivo de microdados. Esses arquivos microdados estão organizados em um formato chamado *fixed width file* (fwf). Basicamente, um arquivo fwf é um no qual cada variável tem uma posição de início e final bem definida e consistente entre as diversas linhas. Por exemplo, a primeira linha do registro DOMICILIO, no arquivo DOMICILIO.txt é:

```
1111031110005400 1111110311112222 3 2112112121 11 272.80666931 372.984515701
```

O que não faz nenhum sentido.

Para interpretar esses dados, precisamos de um dicionário. O IBGE disponibiliza um dicionário de variável, que indica a posição de início, o tamanho, o código, a descrição e as categorias de cada variável, para cada

registro. Esse arquivo está em “./documentacao/Dicionários de variáveis” (está escrito assim). Na aba “Domicílio”, temos as informações necessárias para ler os microdados da POF no R.

Em linhas gerais, precisamos transferir essa informação do dicionário ao R, para que ele crie um df com as colunas corretas. Existem diversas formas de fazer isso. Vou apresentar duas, que consistem em criar um df auxiliar para ler os microdados - esse df auxiliar será chamado de leitores. Pessoalmente, eu acho a segunda mais simples, elegante e segura.

## Fazendo leitores

### Criando um identificador no próprio script

Esse método consiste em criar, no próprio script, um df de identificação. Para o registro DOMICILIO, seria algo na seguinte forma:

```
leitores_domicilio_1 <- tibble(
  posicao_inicial = c(1,3,7,8,
                    17,19,
                    20,21,22,23,25,26,27,28,29,
                    30,31,32,33,34,36,37,38,39,
                    40,41,42,43,44,45,46,47,48,49,
                    50,51,
                    65,
                    79),
  tamanho = c(2,4,1,9,
              2,1,
              1,1,1,2,1,1,1,1,1,
              1,1,1,1,2,1,1,1,1,
              1,1,1,1,1,1,1,1,1,1,
              1,14,
              14,
              1),
  codigo_da_variavel = c('UF','ESTRATO_POF','TIPO_SITUACAO_REG','COD_UPA',
                        'NUM_DOM','V0201',
                        'V0202','V0203','V0204','V0205','V0206','V0207','V0208','V0209','V02101',
                        'V02102','V02103','V02104','V02105','V02111','V02112','V02113','V0212','V0213',
                        'V02141','V02142','V0215','V02161','V02162','V02163','V02164','V0217','V0219','V0220',
                        'V0221','PESO','PESO_FINAL','V6199')
) %>%
  mutate(posicao_final = posicao_inicial + tamanho -1) %>%
  select(posicao_inicial, tamanho, posicao_final, codigo_da_variavel)
```

O problema dessa abordagem é que ela é muito trabalhosa, requer um alto nível de atenção, aumenta desnecessariamente o tamanho do código, é muito propícia para erros e torna muito difícil identificar e corrigir esses erros.

### Criando um leitores.csv

A abordagem que eu uso é a de criar, a partir do arquivo dicionário, um arquivo .csv de leitores.

```
leitores_domicilio_2 <- readxl::read_excel("./documentacao/Dicionários de variáveis.xls",
  sheet = "Domicílio",
```

```

                                skip = 3) %>%
janitor::clean_names() %>%
select(posicao_inicial, tamanho, codigo_da_variavel) %>%
filter(complete.cases()) %>%
mutate(posicao_inicial = as.numeric(posicao_inicial),
       tamanho = as.numeric(tamanho),
       posicao_final = posicao_inicial + tamanho - 1) %>%
select(posicao_inicial, tamanho, posicao_final, codigo_da_variavel)

```

A função `filter(complete.cases())` retira todas as linhas que possuem algum NA. Essa segunda abordagem já pega tudo o que precisamos diretamente do arquivo de dicionário.

```
leitores_domicilio_1
```

```

## # A tibble: 38 x 4
##   posicao_inicial tamanho posicao_final codigo_da_variavel
##         <dbl>   <dbl>         <dbl> <chr>
## 1             1       2             2 UF
## 2             3       4             6 ESTRATO_POF
## 3             7       1             7 TIPO_SITUACAO_REG
## 4             8       9            16 COD_UPA
## 5            17       2            18 NUM_DOM
## 6            19       1            19 V0201
...

```

```
leitores_domicilio_2
```

```

## # A tibble: 38 x 4
##   posicao_inicial tamanho posicao_final codigo_da_variavel
##         <dbl>   <dbl>         <dbl> <chr>
## 1             1       2             2 UF
## 2             3       4             6 ESTRATO_POF
## 3             7       1             7 TIPO_SITUACAO_REG
## 4             8       9            16 COD_UPA
## 5            17       2            18 NUM_DOM
## 6            19       1            19 V0201
...

```

### terminando a configuração

O que precisamos agora é colocar nossa tabela de leitores no formato que o R precisa para ler os microdados. Como eu considero *leitores\_domicilio\_2* a melhor alternativa, vou usar como exemplo.

```

colpos_domicilio <- fwf_positions(start = leitores_domicilio_2$posicao_inicial,
                                end = leitores_domicilio_2$posicao_final,
                                col_names = leitores_domicilio_2$codigo_da_variavel)

colpos_domicilio

```

```

## # A tibble: 38 x 3
##   begin   end col_names

```

```
##      <dbl> <dbl> <chr>
## 1      0      2 UF
## 2      2      6 ESTRATO_POF
## 3      6      7 TIPO_SITUACAO_REG
## 4      7     16 COD_UPA
## 5     16     18 NUM_DOM
## 6     18     19 V0201
...

```

Como é possível ver, as diferenças estão no nome das colunas e na coluna *begin*, que começa do zero.

Por fim, recomendo criar uma pasta “leitores” e colocar nela todas as tabelas leitores em formato csv:

```
# colpos_domicilio %>%
#   write_csv("./leitores/leitores_domicilio.csv")

```

Dois breves comentários. Primeiro, eu sou sempre favorável a manter os nomes de variáveis igual ao IBGE. Isso faz com que, no caso de dúvida, basta olhar no dicionário qual o significado. Se colocarmos outros nomes, precisaremos olhar em dois lugares: um para transformar o nome que demos no nome que o IBGE deu, e aí olhar no dicionário. Segundo, se já sabemos que variáveis vamos usar ou não, podemos simplesmente entrar no arquivo csv e deixar apenas as linhas das variáveis que usaremos. Por um lado, isso pode tornar o processo de leitura dos dados muito mais rápido e melhorar o desempenho do R, porque a tabela carregada será menor. Por outro, pode ser um pouco tedioso refazer todas as etapas se, no meio do caminho, percebermos que esquecemos de algo ou tivermos uma nova ideia. Isso só é possível porque fizemos leitores com posição inicial e posição final de cada variável. Se usássemos apenas o tamanho, precisaríamos ler todas as variáveis.

## Lendo os microdados da POF no R

Agora temos tudo o que precisamos: os microdados em formato *.txt* (na pasta dados, baixada diretamente do site da POF no IBGE) e o arquivo de leitores já no formato certo.

```
colpos_domicilio <- read_csv("./leitores/leitores_domicilio.csv")

pof_domicilio <- read_fwf(file = "./dados/DOMICILIO.txt",
                          col_positions = colpos_domicilio,
                          col_types = cols(.default = col_character()))

```

E pronto! Se tudo estiver certo, já temos o registro Domicílio da POF carregado no R!

```
pof_domicilio

## # A tibble: 57,920 x 38
##   UF      ESTRATO_POF TIPO_SITUACAO_REG COD_UPA   NUM_DOM V0201 V0202 V0203 V0204
##   <chr> <chr>         <chr>         <chr>   <chr>   <chr> <chr> <chr> <chr>
## 1 11    1103         1             110005400 1       1     1     1     1
## 2 11    1103         1             110005400 2       1     1     1     1
## 3 11    1103         1             110005400 4       1     4     1     1
## 4 11    1103         1             110005400 5       1     4     1     1
## 5 11    1103         1             110005400 6       1     1     1     1
## 6 11    1103         1             110005400 7       1     4     1     3
...

```

Para os outros registros, o processo é exatamente o mesmo. Apenas um comentário sobre o comando `col_types = cols(.default = col_character())`. O padrão da função `read_fwf` utiliza uma imputação do formato das colunas, a partir das X primeiras linhas. Isso pode causar problemas. Uma maneira de contornar isso é dando ao R o tipo de cada coluna. A outra é essa que fiz - transformar tudo em *character*. Certamente existem soluções melhores e mais elegantes, mas essa funciona para mim.

Por fim, não precisamos repetir esse processo toda vez. Podemos salvar a POF em *.rds*, que é o formato nativo do R - reduzindo muito o tempo necessário para carregar a POF.

```
pof_domicilio %>%  
  write_rds(file = "./dados/pof_domicilio.rds", compress = "gz")
```

Uma vez feito isso, para carregar os microdados, basta rodar

```
pof_domicilio <- read_rds("./dados/pof_domicilio.rds")
```

## Para fazer solo antes da próxima aula

Criar leitores *.csv* e arquivos *.rds* para os seguintes registros:

- MORADOR
  - 178431 linhas, 57 colunas
- ALUGUEL\_ESTIMADO
  - 48935 linhas, 19 colunas
- CADERNETA\_COLETIVA
  - 789995 linhas, 23 colunas
- DESPESA\_COLETIVA
  - 478572 linhas, 27 colunas
- DESPESA\_INDIVIDUAL
  - 1836032 linhas, 25 colunas
- OUTROS\_RENDIMENTOS
  - 206108 linhas, 22 colunas
- RENDIMENTO\_TRABALHO
  - 97075 linhas, 37 colunas

## Exercícios de revisão do R

1. Temos dois arquivos *.csv* distintos e queremos fazer um *left\_join*. Qual a coluna chave? Faça com que as colunas do arquivo *time\_regiao.csv* estejam antes das colunas do arquivo *classif.csv* no df final. Salve essa nova tabela como *classif\_final.csv*
2. Crie colunas com o número de pontos (vitórias valem 3 pontos, empates valem 1 e derrotas valem 0), o número de jogos e o saldo de gols (gols pró menos gols contra) de cada time.
3. O critério de desempate é número de pontos, número de vitórias e saldo de gols. Se fosse número de pontos, saldo de gols e número de vitórias, a classificação mudaria? Dê um exemplo.

4. Em dois dfs distintos, apresente o nome, a pontuação e o número de gols dos 3 primeiros e os 4 últimos colocados (dica: funções *head* e *tail*).
5. Filtre os times do Sul.
6. Faça 2 df-síntese. O primeiro com o total, a média e a mediana dos pontos e o segundo com as mesmas informações, agrupado por região.

#### **dicas**

1. A coluna chave é uma que está presente nos dois dfs e pode ser usada para juntá-los. A ordem do *left\_join* é importante! A ordenação também pode ser feita com *select* (mas mais trabalhoso).
2. Usar *mutate*. Pense em bons nomes para as colunas criadas.
3. A função *arrange* pode envolver mais de uma coluna. A ordem importa!
4. Preste atenção no número de linhas e colunas! Veja a documentação com *?head* e *?tail*.
5. Pense na diferença entre *=* e *==*.
6. Lembre que você pode criar várias colunas com *summarise*.