Chapter 4 The tidyverse

Up to now we have been manipulating vectors by reordering and subsetting them through indexing. However, once we start more advanced analyses, the preferred unit for data storage is not the vector but the data frame. In this chapter we learn to work directly with data frames, which greatly facilitate the organization of information. We will be using data frames for the majority of this book. We will focus on a specific data format referred to as tidy and on specific collection of packages that are particularly helpful for working with tidy data referred to as the tidyverse.

We can load all the tidyverse packages at once by installing and loading the tidyverse package:

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.4     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   2.0.1     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

We will learn how to implement the tidyverse approach throughout the book, but before delving into the details, in this chapter we introduce some of the most widely used tidyverse functionality, starting with the dplyr package for manipulating data frames and the purrr package for working with functions. Note that the tidyverse also includes a graphing package, ggplot2, which we introduce later in Chapter 7 in the Data Visualization part of the book; the readr package discussed in Chapter 5; and many others. In this chapter, we first introduce the concept of tidy data and then demonstrate how we use the tidyverse to work with data frames in this format.

4.1 Tidy data

We say that a data table is in tidy format if each row represents one observation and columns represent the different variables available for each of these observations. The murders dataset is an example of a tidy data frame.

#> state abb region population total #> 1 Alabama AL South 4779736 135 #> 2 Alaska AK West 710231 19 #> 3 Arizona AZ West 6392017 232 #> 4 Arkansas AR South 2915918 93 #> 5 California CA West 37253956 1257 #> 6 Colorado CO West 5029196 65

```
library(dslabs)
head(murders)
```

```
##        state abb region population total
## 1    Alabama  AL  South    4779736   135
## 2     Alaska  AK   West     710231    19
## 3    Arizona  AZ   West    6392017   232
## 4   Arkansas  AR  South    2915918    93
## 5 California  CA   West   37253956  1257
## 6   Colorado  CO   West    5029196    65
```

Each row represent a state with each of the five columns providing a different variable related to these states: name, abbreviation, region, population, and total murders.

To see how the same information can be provided in different formats, consider the following example:

#> country year fertility #> 1 Germany 1960 2.41 #> 2 South Korea 1960 6.16 #> 3 Germany 1961 2.44 #> 4 South Korea 1961 5.99 #> 5 Germany 1962 2.47 #> 6 South Korea 1962 5.79 This tidy dataset provides fertility rates for two countries across the years. This is a tidy dataset because each row presents one observation with the three variables being country, year, and fertility rate. However, this dataset originally came in another format and was reshaped for the dslabs package. Originally, the data was in the following format:

#> country 1960 1961 1962 #> 1 Germany 2.41 2.44 2.47 #> 2 South Korea 6.16 5.99 5.79 The same information is provided, but there are two important differences in the format: 1) each row includes several observations and 2) one of the variables, year, is stored in the header. For the tidyverse packages to be optimally used, data need to be reshaped into tidy format, which you will learn to do in the Data Wrangling part of the book. Until then, we will use example datasets that are already in tidy format.

Although not immediately obvious, as you go through the book you will start to appreciate the advantages of working in a framework in which functions use tidy formats for both inputs and outputs. You will see how this permits the data analyst to focus on more important aspects of the analysis rather than the format of the data.

##4.2 Exercises 1. Examine the built-in dataset co2. Which of the following is true:

```
data("co2")
View(co2)
co2
```

```
##         Jan    Feb    Mar    Apr    May    Jun    Jul    Aug    Sep    Oct
## 1959 315.42 316.31 316.50 317.56 318.13 318.00 316.39 314.65 313.68 313.18
## 1960 316.27 316.81 317.42 318.87 319.87 319.43 318.01 315.74 314.00 313.68
## 1961 316.73 317.54 318.38 319.31 320.42 319.61 318.42 316.63 314.83 315.16
## 1962 317.78 318.40 319.53 320.42 320.85 320.45 319.45 317.25 316.11 315.27
## 1963 318.58 318.92 319.70 321.22 322.08 321.31 319.58 317.61 316.05 315.83
## 1964 319.41 320.07 320.74 321.40 322.06 321.73 320.27 318.54 316.54 316.71
## 1965 319.27 320.28 320.73 321.97 322.00 321.71 321.05 318.71 317.66 317.14
## 1966 320.46 321.43 322.23 323.54 323.91 323.59 322.24 320.20 318.48 317.94
## 1967 322.17 322.34 322.88 324.25 324.83 323.93 322.38 320.76 319.10 319.24
## 1968 322.40 322.99 323.73 324.86 325.40 325.20 323.98 321.95 320.18 320.09
## 1969 323.83 324.26 325.47 326.50 327.21 326.54 325.72 323.50 322.22 321.62
## 1970 324.89 325.82 326.77 327.97 327.91 327.50 326.18 324.53 322.93 322.90
## 1971 326.01 326.51 327.01 327.62 328.76 328.40 327.20 325.27 323.20 323.40
## 1972 326.60 327.47 327.58 329.56 329.90 328.92 327.88 326.16 324.68 325.04
## 1973 328.37 329.40 330.14 331.33 332.31 331.90 330.70 329.15 327.35 327.02
## 1974 329.18 330.55 331.32 332.48 332.92 332.08 331.01 329.23 327.27 327.21
## 1975 330.23 331.25 331.87 333.14 333.80 333.43 331.73 329.90 328.40 328.17
## 1976 331.58 332.39 333.33 334.41 334.71 334.17 332.89 330.77 329.14 328.78
## 1977 332.75 333.24 334.53 335.90 336.57 336.10 334.76 332.59 331.42 330.98
## 1978 334.80 335.22 336.47 337.59 337.84 337.72 336.37 334.51 332.60 332.38
## 1979 336.05 336.59 337.79 338.71 339.30 339.12 337.56 335.92 333.75 333.70
## 1980 337.84 338.19 339.91 340.60 341.29 341.00 339.39 337.43 335.72 335.84
## 1981 339.06 340.30 341.21 342.33 342.74 342.08 340.32 338.26 336.52 336.68
## 1982 340.57 341.44 342.53 343.39 343.96 343.18 341.88 339.65 337.81 337.69
## 1983 341.20 342.35 342.93 344.77 345.58 345.14 343.81 342.21 339.69 339.82
## 1984 343.52 344.33 345.11 346.88 347.25 346.62 345.22 343.11 340.90 341.18
## 1985 344.79 345.82 347.25 348.17 348.74 348.07 346.38 344.51 342.92 342.62
```

```
## 1986 346.11 346.78 347.68 349.37 350.03 349.37 347.76 345.73 344.68 343.99
## 1987 347.84 348.29 349.23 350.80 351.66 351.07 349.33 347.92 346.27 346.18
## 1988 350.25 351.54 352.05 353.41 354.04 353.62 352.22 350.27 348.55 348.72
## 1989 352.60 352.92 353.53 355.26 355.52 354.97 353.75 351.52 349.64 349.83
## 1990 353.50 354.55 355.23 356.04 357.00 356.07 354.67 352.76 350.82 351.04
## 1991 354.59 355.63 357.03 358.48 359.22 358.12 356.06 353.92 352.05 352.11
## 1992 355.88 356.63 357.72 359.07 359.58 359.17 356.94 354.92 352.94 353.23
## 1993 356.63 357.10 358.32 359.41 360.23 359.55 357.53 355.48 353.67 353.95
## 1994 358.34 358.89 359.95 361.25 361.67 360.94 359.55 357.49 355.84 356.00
## 1995 359.98 361.03 361.66 363.48 363.82 363.30 361.94 359.50 358.11 357.80
## 1996 362.09 363.29 364.06 364.76 365.45 365.01 363.70 361.54 359.51 359.65
## 1997 363.23 364.06 364.61 366.40 366.84 365.68 364.52 362.57 360.24 360.83
##         Nov    Dec
## 1959 314.66 315.43
## 1960 314.84 316.03
## 1961 315.94 316.85
## 1962 316.53 317.53
## 1963 316.91 318.20
## 1964 317.53 318.55
## 1965 318.70 319.25
## 1966 319.63 320.87
## 1967 320.56 321.80
## 1968 321.16 322.74
## 1969 322.69 323.95
## 1970 323.85 324.96
## 1971 324.63 325.85
## 1972 326.34 327.39
## 1973 327.99 328.48
## 1974 328.29 329.41
## 1975 329.32 330.59
## 1976 330.14 331.52
## 1977 332.24 333.68
## 1978 333.75 334.78
## 1979 335.12 336.56
## 1980 336.93 338.04
## 1981 338.19 339.44
## 1982 339.09 340.32
## 1983 340.98 342.82
## 1984 342.80 344.04
## 1985 344.06 345.38
## 1986 345.48 346.72
## 1987 347.64 348.78
## 1988 349.91 351.18
## 1989 351.14 352.37
## 1990 352.69 354.07
## 1991 353.64 354.89
## 1992 354.09 355.33
## 1993 355.30 356.78
## 1994 357.59 359.05
## 1995 359.61 360.74
## 1996 360.80 362.38
## 1997 362.49 364.34
```

Answer(A)

2. Examine the built-in dataset ChickWeight. Which of the following is true:

```
head(ChickWeight)
```

```
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
## 3     59    4     1    1
## 4     64    6     1    1
## 5     76    8     1    1
## 6     93   10     1    1
```

Answer(B)

3. Examine the built-in dataset BOD. Which of the following is true:

```
head(BOD)
```

```
##   Time demand
## 1    1    8.3
## 2    2   10.3
## 3    3   19.0
## 4    4   16.0
## 5    5   15.6
## 6    7   19.8
```

Answer(C)

4. Which of the following built-in datasets is tidy (you can pick more than one):

```
head(BJsales)
```

```
## [1] 200.1 199.5 199.4 198.9 199.0 200.2
```

```
View(BJsales)
```

```
head(EuStockMarkets)
```

```
##          DAX    SMI    CAC   FTSE
## [1,] 1628.75 1678.1 1772.8 2443.6
## [2,] 1613.63 1688.5 1750.5 2460.2
## [3,] 1606.51 1678.6 1718.0 2448.2
## [4,] 1621.04 1684.1 1708.1 2470.4
## [5,] 1618.16 1686.6 1723.1 2484.7
## [6,] 1610.61 1671.6 1714.3 2466.8
```

```
View(EuStockMarkets)
```

```
head(DNase)
```

```
##   Run       conc density
## 1   1 0.04882812   0.017
## 2   1 0.04882812   0.018
## 3   1 0.19531250   0.121
## 4   1 0.19531250   0.124
## 5   1 0.39062500   0.206
## 6   1 0.39062500   0.215
```

```
head(Formaldehyde)
```

```
##   carb optden
```

```
## 1  0.1  0.086
## 2  0.3  0.269
## 3  0.5  0.446
## 4  0.6  0.538
## 5  0.7  0.626
## 6  0.9  0.782
```

```
head(Orange)
```

```
##   Tree  age circumference
## 1    1  118            30
## 2    1  484            58
## 3    1  664            87
## 4    1 1004           115
## 5    1 1231           120
## 6    1 1372           142
```

```
head(UCBAdmissions)
```

```
## , , Dept = A
##
##           Gender
## Admit      Male Female
##   Admitted  512     89
##   Rejected  313     19
##
## , , Dept = B
##
##           Gender
## Admit      Male Female
##   Admitted  353     17
##   Rejected  207      8
##
## , , Dept = C
##
##           Gender
## Admit      Male Female
##   Admitted  120    202
##   Rejected  205    391
##
## , , Dept = D
##
##           Gender
## Admit      Male Female
##   Admitted  138    131
##   Rejected  279    244
##
## , , Dept = E
##
##           Gender
## Admit      Male Female
##   Admitted   53     94
##   Rejected  138    299
##
## , , Dept = F
```

```
##
##           Gender
## Admit      Male Female
##   Admitted   22     24
##   Rejected  351    317
```

```
View(UCBAdmissions)
```

Answer. b,c,d,e,f

4.3 Manipulating data frames

The dplyr package from the tidyverse introduces functions that perform some of the most common operations when working with data frames and uses names for these functions that are relatively easy to remember. For instance, to change the data table by adding a new column, we use mutate. To filter the data table to a subset of rows, we use filter. Finally, to subset the data by selecting specific columns, we use select.

4.3.1 Adding a column with mutate

We want all the necessary information for our analysis to be included in the data table. So the first task is to add the murder rates to our murders data frame. The function mutate takes the data frame as a first argument and the name and values of the variable as a second argument using the convention name = values. So, to add murder rates, we use:

```
library(dslabs)
data("murders")
murders <- mutate(murders, rate = total/ population * 100000)
```

Notice that here we used total and population inside the function, which are objects that are not defined in our workspace. But why don't we get an error?

This is one of dplyr's main features. Functions in this package, such as mutate, know to look for variables in the data frame provided in the first argument. In the call to mutate above, total will have the values in murders$total. This approach makes the code much more readable.

We can see that the new column is added:

```
head(murders)
```

```
##        state abb region population total     rate
## 1    Alabama  AL  South    4779736   135 2.824424
## 2     Alaska  AK   West     710231    19 2.675186
## 3    Arizona  AZ   West    6392017   232 3.629527
## 4   Arkansas  AR  South    2915918    93 3.189390
## 5 California  CA   West   37253956  1257 3.374138
## 6   Colorado  CO   West    5029196    65 1.292453
```

Although we have overwritten the original murders object, this does not change the object that loaded with data(murders). If we load the murders data again, the original will overwrite our mutated version.

4.3.2 Subsetting with filter

Now suppose that we want to filter the data table to only show the entries for which the murder rate is lower than 0.71. To do this we use the filter function, which takes the data table as the first argument and then the conditional statement as the second. Like mutate, we can use the unquoted variable names from murders inside the function and it will know we mean the columns and not objects in the workspace.

```
filter(murders, rate <= 0.71)
```

```
##           state abb        region population total      rate
## 1        Hawaii  HI          West    1360301     7 0.5145920
## 2          Iowa  IA North Central    3046355    21 0.6893484
## 3 New Hampshire  NH     Northeast    1316470     5 0.3798036
## 4  North Dakota  ND North Central     672591     4 0.5947151
## 5       Vermont  VT     Northeast     625741     2 0.3196211
```

4.3.3 Selecting columns with select

Although our data table only has six columns, some data tables include hundreds. If we want to view just a few, we can use the dplyr select function. In the code below we select three columns, assign this to a new object and then filter the new object:

```
new_table <- select(murders, state, region, rate)
filter(new_table, rate <= 0.71)
```

```
##           state        region      rate
## 1        Hawaii          West 0.5145920
## 2          Iowa North Central 0.6893484
## 3 New Hampshire     Northeast 0.3798036
## 4  North Dakota North Central 0.5947151
## 5       Vermont     Northeast 0.3196211
```

In the call to select, the first argument murders is an object, but state, region, and rate are variable names.

4.4 Exercises

1. Load the dplyr package and the murders dataset.

```
library(dplyr)
library(dslabs)
data(murders)
```

You can add columns using the dplyr function mutate. This function is aware of the column names and inside the function you can call them unquoted:

```
murders <- mutate(murders, population_in_millions = population / 10^6)
```

We can write population rather than murders$population. The function mutate knows we are grabbing columns from murders.

Use the function mutate to add a murders column named rate with the per 100,000 murder rate as in the example code above. Make sure you redefine murders as done in the example code above (murders <- [your code]) so we can keep using this variable.

```
murders <- mutate(murders, rate = total / population *100000)
```

2. If rank(x) gives you the ranks of x from lowest to highest, rank(-x) gives you the ranks from highest to lowest. Use the function mutate to add a column rank containing the rank, from highest to lowest murder rate. Make sure you redefine murders so we can keep using this variable.

```
murders <- mutate(murders, rank = rank(-rate))
```

3. With dplyr, we can use select to show only certain columns. For example, with this code we would only show the states and population sizes:

```
select(murders, state, population)  %>% head()
```

```
##        state population
## 1    Alabama    4779736
```

```
## 2     Alaska    710231
## 3    Arizona   6392017
## 4   Arkansas   2915918
## 5 California  37253956
## 6   Colorado   5029196
```

Use select to show the state names and abbreviations in murders. Do not redefine murders, just show the results.

```
select(murders, state, abb) %>% head()
```

```
##        state abb
## 1    Alabama  AL
## 2     Alaska  AK
## 3    Arizona  AZ
## 4   Arkansas  AR
## 5 California  CA
## 6   Colorado  CO
```

4. The dplyr function filter is used to choose specific rows of the data frame to keep. Unlike select which is for columns, filter is for rows. For example, you can show just the New York row like this:

```
filter(murders, state == "New York")
```

```
##      state abb    region population total population_in_millions    rate rank
## 1 New York  NY Northeast   19378102   517                19.3781 2.66796   29
```

You can use other logical vectors to filter rows.

Use filter to show the top 5 states with the highest murder rates. After we add murder rate and rank, do not change the murders dataset, just show the result. Remember that you can filter based on the rank column.

```
dplyr::filter(murders, rank <= 5)
```

```
##                  state abb        region population total
## 1 District of Columbia  DC         South     601723    99
## 2            Louisiana  LA         South    4533372   351
## 3             Maryland  MD         South    5773552   293
## 4             Missouri  MO North Central    5988927   321
## 5       South Carolina  SC         South    4625364   207
##   population_in_millions      rate rank
## 1               0.601723 16.452753    1
## 2               4.533372  7.742581    2
## 3               5.773552  5.074866    4
## 4               5.988927  5.359892    3
## 5               4.625364  4.475323    5
```

5. We can remove rows using the != operator. For example, to remove Florida, we would do this:

```
no_florida <- filter(murders, state != "Florida")
```

Create a new data frame called no_south that removes states from the South region. How many states are in this category? You can use the function nrow for this.

```
no_south <- filter(murders, region != "South")
nrow(no_south)
```

```
## [1] 34
```

6. We can also use %in% to filter with dplyr. You can therefore see the data from New York and Texas like this:

```
filter(murders, state %in% c("New York", "Texas"))
```

```
##      state abb    region population total population_in_millions    rate rank
## 1 New York  NY Northeast   19378102   517               19.37810 2.66796   29
## 2    Texas  TX     South   25145561   805               25.14556 3.20136   16
```

Create a new data frame called murders_nw with only the states from the Northeast and the West. How many states are in this category?

```
murders_nw <- filter(murders, region %in% c("Northeast", "West"))
nrow(murders_nw)
```

```
## [1] 22
```

7. Suppose you want to live in the Northeast or West and want the murder rate to be less than 1. We want to see the data for the states satisfying these options. Note that you can use logical operators with filter. Here is an example in which we filter to keep only small states in the Northeast region.

```
filter(murders, population < 5000000 & region == "Northeast")
```

```
##           state abb    region population total population_in_millions      rate
## 1   Connecticut  CT Northeast    3574097    97               3.574097 2.7139722
## 2         Maine  ME Northeast    1328361    11               1.328361 0.8280881
## 3 New Hampshire  NH Northeast    1316470     5               1.316470 0.3798036
## 4  Rhode Island  RI Northeast    1052567    16               1.052567 1.5200933
## 5       Vermont  VT Northeast     625741     2               0.625741 0.3196211
##   rank
## 1   25
## 2   44
## 3   50
## 4   35
## 5   51
```

Make sure murders has been defined with rate and rank and still has all states. Create a table called my_states that contains rows for states satisfying both the conditions: it is in the Northeast or West and the murder rate is less than 1. Use select to show only the state name, the rate and the rank.

```
my_state <- dplyr::filter(murders, region %in% c("Northeast", "West") & rate <= 1)
select(my_state, state, rate, rank)
```

```
##           state      rate rank
## 1        Hawaii 0.5145920   49
## 2         Idaho 0.7655102   46
## 3         Maine 0.8280881   44
## 4 New Hampshire 0.3798036   50
## 5        Oregon 0.9396843   42
## 6          Utah 0.7959810   45
## 7       Vermont 0.3196211   51
## 8       Wyoming 0.8871131   43
```

4.5. The pipe : %>%

With dplyr we can perform a series of operations, for example select and then filter, by sending the results of one function to another using what is called the pipe operator: %>%. Some details are included below.

We wrote code above to show three variables (state, region, rate) for states that have murder rates below 0.71. To do this, we defined the intermediate object new_table. In dplyr we can write code that looks more like a description of what we want to do without intermediate objects:

original data → select → filter

For such an operation, we can use the pipe %>%. The code looks like this:

```
murders %>% select(state, region, rate) %>% filter(rate <= 0.71)
```

```
##            state        region      rate
## 1         Hawaii          West 0.5145920
## 2           Iowa North Central 0.6893484
## 3 New Hampshire      Northeast 0.3798036
## 4  North Dakota North Central 0.5947151
## 5        Vermont     Northeast 0.3196211
```

This line of code is equivalent to the two lines of code above. What is going on here?

In general, the pipe sends the result of the left side of the pipe to be the first argument of the function on the right side of the pipe. Here is a very simple example:

```
16 %>% sqrt()
```

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

We can continue to pipe values along:

```
16 %>% sqrt() %>% log2()
```

```
## [1] 2
```

The above statement is equivalent to log2(sqrt(16)).

Remember that the pipe sends values to the first argument, so we can define other arguments as if the first argument is already defined:

```
16 %>% sqrt() %>% log(base = 2)
```

```
## [1] 2
```

Therefore, when using the pipe with data frames and dplyr, we no longer need to specify the required first argument since the dplyr functions we have described all take the data as the first argument. In the code we wrote:

```
murders %>% select(state, region, rate) %>% filter(rate <= 0.71)
```

```
##            state        region      rate
## 1         Hawaii          West 0.5145920
## 2           Iowa North Central 0.6893484
## 3 New Hampshire      Northeast 0.3798036
## 4  North Dakota North Central 0.5947151
## 5        Vermont     Northeast 0.3196211
```

murders is the first argument of the select function, and the new data frame (formerly new_table) is the first argument of the filter function.

Note that the pipe works well with functions where the first argument is the input data. Functions in tidyverse packages like dplyr have this format and can be used easily with the pipe.

4.6 Exercises

1. The pipe %>% can be used to perform operations sequentially without having to define intermediate objects. Start by redefining murder to include rate and rank.

```
murders <- dplyr::mutate(murders, rate =  total / population * 100000,
                 rank = rank(-rate))
```

In the solution to the previous exercise, we did the following:

```
my_states <- dplyr::filter(murders, region %in% c("Northeast", "West") &
                      rate < 1)

select(my_states, state, rate, rank)
```

```
##           state      rate rank
## 1        Hawaii 0.5145920   49
## 2         Idaho 0.7655102   46
## 3         Maine 0.8280881   44
## 4 New Hampshire 0.3798036   50
## 5        Oregon 0.9396843   42
## 6          Utah 0.7959810   45
## 7       Vermont 0.3196211   51
## 8       Wyoming 0.8871131   43
```

The pipe %>% permits us to perform both operations sequentially without having to define an intermediate variable my_states. We therefore could have mutated and selected in the same line like this:

```
mutate(murders, rate =  total / population * 100000,
       rank = rank(-rate)) %>%
  select(state, rate, rank)
```

```
##                   state       rate rank
## 1               Alabama  2.8244238   23
## 2                Alaska  2.6751860   27
## 3               Arizona  3.6295273   10
## 4              Arkansas  3.1893901   17
## 5            California  3.3741383   14
## 6              Colorado  1.2924531   38
## 7           Connecticut  2.7139722   25
## 8              Delaware  4.2319369    6
## 9  District of Columbia 16.4527532    1
## 10              Florida  3.3980688   13
## 11              Georgia  3.7903226    9
## 12               Hawaii  0.5145920   49
## 13                Idaho  0.7655102   46
## 14             Illinois  2.8369608   22
## 15              Indiana  2.1900730   31
## 16                 Iowa  0.6893484   47
## 17               Kansas  2.2081106   30
## 18             Kentucky  2.6732010   28
## 19            Louisiana  7.7425810    2
## 20                Maine  0.8280881   44
## 21             Maryland  5.0748655    4
## 22        Massachusetts  1.8021791   32
## 23             Michigan  4.1786225    7
## 24            Minnesota  0.9992600   40
## 25          Mississippi  4.0440846    8
## 26             Missouri  5.3598917    3
## 27              Montana  1.2128379   39
## 28             Nebraska  1.7521372   33
## 29               Nevada  3.1104763   19
## 30        New Hampshire  0.3798036   50
## 31           New Jersey  2.7980319   24
```

```
## 32           New Mexico  3.2537239   15
## 33             New York  2.6679599   29
## 34       North Carolina  2.9993237   20
## 35         North Dakota  0.5947151   48
## 36                 Ohio  2.6871225   26
## 37             Oklahoma  2.9589340   21
## 38               Oregon  0.9396843   42
## 39         Pennsylvania  3.5977513   11
## 40         Rhode Island  1.5200933   35
## 41       South Carolina  4.4753235    5
## 42         South Dakota  0.9825837   41
## 43            Tennessee  3.4509357   12
## 44                Texas  3.2013603   16
## 45                 Utah  0.7959810   45
## 46              Vermont  0.3196211   51
## 47             Virginia  3.1246001   18
## 48           Washington  1.3829942   37
## 49        West Virginia  1.4571013   36
## 50            Wisconsin  1.7056487   34
## 51              Wyoming  0.8871131   43
```

Notice that select no longer has a data frame as the first argument. The first argument is assumed to be the result of the operation conducted right before the %>%.

Repeat the previous exercise, but now instead of creating a new object, show the result and only include the state, rate, and rank columns. Use a pipe %>% to do this in just one line.

```
murders %>% filter(region %in% c("Northeast", "West") & rate < 1) %>% select(state, rate, rank)
```

```
##           state      rate rank
## 1        Hawaii 0.5145920   49
## 2         Idaho 0.7655102   46
## 3         Maine 0.8280881   44
## 4 New Hampshire 0.3798036   50
## 5        Oregon 0.9396843   42
## 6          Utah 0.7959810   45
## 7       Vermont 0.3196211   51
## 8       Wyoming 0.8871131   43
```

2. Reset murders to the original table by using data(murders). Use a pipe to create a new data frame called my_states that considers only states in the Northeast or West which have a murder rate lower than 1, and contains only the state, rate and rank columns. The pipe should also have four components separated by three %>%. The code should look something like this:

```
#my_states <- murders %>%
```

mutate SOMETHING %>% filter SOMETHING %>% select SOMETHING

```
data(murders)
my_states <- murders %>% mutate(rate = total / population * 100000, rank = rank(-rate)) %>% filter(regi
my_states
```

```
##           state      rate rank
## 1        Hawaii 0.5145920   49
## 2         Idaho 0.7655102   46
## 3         Maine 0.8280881   44
## 4 New Hampshire 0.3798036   50
## 5        Oregon 0.9396843   42
```

```
## 6          Utah 0.7959810   45
## 7       Vermont 0.3196211   51
## 8       Wyoming 0.8871131   43
```