04 The tidyverse

Soyoung Park

Pusan National University
Department of Statistics

Tidyverse

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

```
library(tidyverse)
```

Tidy data

When are data **tidy**? When they satisfy these three conditions:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Every other arrangement of data is called **"messy."**

Data often is is a spreadsheet format, but

there's different ways of encoding the same information

```
##Option #1

##           name treatmenta treatmentb
## 1   John Smith         NA         18
## 2     Jane Doe          4          1
## 3 Mary Johnson          6          7



##Option #2

##   treatment John.Smith Jane.Doe Mary.Johnson
## 1         a         NA        4            6
## 2         b         18        1            7
```

Neither #1 nor #2 are "clean" versions of the data: observed information is

Sources of Messiness

1. Column headers are values, not variable names. e.g. *treatmenta*, *treatmentb*
2. Multiple variables are stored in one column. e.g. *Fall 2015, Spring 2016 or "1301 8th St SE, Orange City, Iowa 51041 (42.99755, -96.04149)", "2102 Durant, Harlan, Iowa 51537 (41.65672, -95.33780)"*
3. Multiple observational units are stored in the same table.
4. A single observational unit is stored in multiple tables.

Tidy data

While both options may look quite organized, neither corresponds to tidy data. In both cases, Wickham's rules 1 and 2 are violated.

```
## data 1
#week    city_A    city_B    city_C
#  1         14        18        23
#  2         15        21        24
#  3         12        25        23
#  4         13        17        25
# the variable temperature appears in three columns

## data 2
#week         1        2        3        4      ...
#city_A      14       15       12       13
#city_B      18       21       25       17
#city_C      23       24       23       25
#multiple variables appear in each column and multiple observ
```

Tidy version of data 1

```
#week    city    temperature
#  1       A              14
#  1       B              18
#  1       C              23
#  2       A              15
#  2       B              21
#  2       C              24
#    ...
```

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.

```
## Warning: package 'dslabs' was built under R version 4.0.5
```

```
##        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
```

Tidy data

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 is a tidy dataset because each row presents one observation with the three variables being country, year, and fertility rate.

Tidy data

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

2) one of the variables, year, is stored in the header.

Exercises

1. Examine the bulit-in dataset co2. Which of the following is true:

a. co2 is tidy data: it has one year for each row.
b. co2 is not tidy: we need at least one column with a character vector.
c. co2 is not tidy: it is a matrix instead of a data frame.
d. co2 is not tidy: to be tidy we would have to wrangle it to have three columns (year, month and value), then each co2 observation would have a row.

Exercises

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

a. `ChickWeight` is not tidy: each chick has more than one row.
b. `ChickWeight` is tidy: each observation (a weight) is represented by one row. The chick from which this measurement came is one of the variables.
c. `ChickWeight` is not tidy: we are missing the year column.
d. `ChickWeight` is tidy: it is stored in a data frame.

Exercises

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

a. `BOD` is not tidy: it only has six rows.
b. `BOD` is not tidy: the first column is just an index.
c. `BOD` is tidy: each row is an observation with two values (time and demand)
d. `BOD` is tidy: all small datasets are tidy by definition.

Exercises

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

a. `BJsales`
b. `EuStockMarkets`
c. `DNase`
d. `Formaldehyde`
e. `Orange`
f. `UCBAdmissions`

Adding a column with `mutate`

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)
head(murders,5)
```

```
##        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
```

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.

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

```
##              state abb        region population total       rat
## 1           Hawaii  HI          West    1360301     7 0.514592
## 2             Iowa  IA North Central    3046355    21 0.689348
## 3 New Hampshire  NH     Northeast    1316470     5 0.379803
## 4  North Dakota  ND North Central     672591     4 0.594715
## 5          Vermont  VT     Northeast     625741     2 0.319621
```

Selecting columns with `select`

If we want to view just a few, we can use the **dplyr** `select` function.

```
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
```

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`:

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

Use the function `mutate` to add a murders column named rate with the per 100,000 murder rate as in the example code above. And redefine `murders` as done in the example code above (murders <- [your code]).

Exercises

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.

Exercises

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()
```

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

Exercises

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")
```

You can use other logical vectors to filter rows.

Use `filter` to show the top 5 states with the highest murder rates. Remember that you can filter based on the `rank` column.

Exercises

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.

Exercises

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"))
```

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?

Exercises

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. Here is an example in which we `filter` to keep only small states in the Northeast region.

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

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.

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:

$$\text{original data} \rightarrow \text{select} \rightarrow \text{filter}$$

The pipe: %>%

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

```
murders %>% select(state, region, rate) %>% filter(rate <= 0.7
```

```
##           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?

The pipe: %>%

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))`.

The pipe: %>%

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 <= 0.7
```

Exercises

1. 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.

Exercises

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
```

Summarizing data

An important part of exploratory data analysis is summarizing data. The average and standard deviation are two examples of widely used summary statistics. More informative summaries can often be achieved by first splitting data into groups. In this section, we cover two new **dplyr** verbs that make these computations easier: `summarize` and `group_by`.

summarize

We start with a simple example based on heights. The `heights` dataset includes heights and sex reported by students in an in-class survey.

```
library(dplyr)
library(dslabs)
data(heights)
```

summarize

The following code computes the average and standard deviation for females:

```
s <- heights %>%
  filter(sex == "Female") %>%
  summarize(average = mean(height), standard_deviation = sd(he
s
```

```
##    average standard_deviation
## 1 64.93942           3.760656
```

summarize

For another example of how we can use the `summarize` function, let's compute the average murder rate for the United States.

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

Remember that the US murder rate is **not** the average of the state murder rates:

```
summarize(murders, mean(rate))
```

```
##   mean(rate)
## 1   2.779125
```

summarize

This is because in the computation above the small states are given the same weight as the large ones. The US murder rate is the total number of murders in the US divided by the total US population. So the correct computation is:

```
us_murder_rate <- murders %>%
  summarize(rate = sum(total) / sum(population) * 100000)
us_murder_rate
```

```
##       rate
## 1 3.034555
```

This computation counts larger states proportionally to their size which results in a larger value.

Multiple summaries

Suppose we want three summaries from the same variable such as the median, minimum, and maximum heights. We can use `summarize` like this:

```
heights %>%
  filter(sex == "Female") %>%
  summarize(median_min_max = quantile(height, c(0.5, 0, 1)))
```

```
##   median_min_max
## 1       64.98031
## 2       51.00000
## 3       79.00000
```

Multiple summaries

However, notice that the summaries are returned in a row each. To obtain the results in different columns, we have to define a function that returns a data frame like this:

```
median_min_max <- function(x){
  qs <- quantile(x, c(0.5, 0, 1))
  data.frame(median = qs[1], minimum = qs[2], maximum = qs[3])
}
heights %>%
  filter(sex == "Female") %>%
  summarize(median_min_max(height))
```

```
##     median minimum maximum
## 1 64.98031      51      79
```

Group then summarize with group_by

We may want to compute the average and standard deviation for men's and women's heights separately. The group_by function helps us do this.

```
heights %>% group_by(sex)
```

```
## # A tibble: 1,050 x 2
## # Groups:   sex [2]
##    sex    height
##    <fct>   <dbl>
##  1 Male       75
##  2 Male       70
##  3 Male       68
##  4 Male       74
##  5 Male       61
##  6 Female     65
##  7 Female     66
```

Group then summarize with group_by

When we summarize the data after grouping, this is what happens:

```
heights %>%
  group_by(sex) %>%
  summarize(average = mean(height),
            standard_deviation = sd(height))
```

```
## # A tibble: 2 x 3
##   sex    average standard_deviation
## * <fct>    <dbl>              <dbl>
## 1 Female    64.9               3.76
## 2 Male      69.3               3.61
```

The summarize function applies the summarization to each group separately.

Group then summarize with `group_by`

For another example, let's compute the median, minimum, and maximum murder rate in the four regions of the country using the `median_min_max` defined above:

```
murders %>%
  group_by(region) %>%
  summarize(median_min_max(rate))
```

```
## # A tibble: 4 x 4
##   region        median minimum maximum
## * <fct>          <dbl>   <dbl>   <dbl>
## 1 Northeast       1.80   0.320    3.60
## 2 South           3.40   1.46    16.5
## 3 North Central   1.97   0.595    5.36
## 4 West            1.29   0.515    3.63
```

pull

The us_murder_rate object defined above represents just one number. Yet we are storing it in a data frame:

```
class(us_murder_rate)
```

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

since, as most **dplyr** functions, summarize always returns a data frame.

pull

This might be problematic if we want to use this result with functions that require a numeric value. Here we show a useful trick for accessing values stored in data when using pipes:

```
us_murder_rate %>% pull(rate)
```

```
## [1] 3.034555
```

This returns the value in the rate column of `us_murder_rate` making it equivalent to `us_murder_rate$rate`.

pull

To get a number from the original data table with one line of code we can type:

```
us_murder_rate <- murders %>%
  summarize(rate = sum(total) / sum(population) * 100000) %>%
  pull(rate)

us_murder_rate
```

```
## [1] 3.034555
```

which is now a numeric:

```
class(us_murder_rate)
```

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

Sorting data frames

We know about the order and sort function, but for ordering entire tables, the **dplyr** function arrange is useful. For example, here we order the states by population size:

```
murders %>%
  arrange(population) %>%
  head()
```

```
##                  state abb        region population total
## 1              Wyoming  WY          West     563626     5
## 2 District of Columbia  DC         South     601723    99 1
## 3              Vermont  VT     Northeast     625741     2
## 4         North Dakota  ND North Central     672591     4
## 5               Alaska  AK          West     710231    19
## 6         South Dakota  SD North Central     814180     8
##   population_in_millions
## 1               0.563626
```

Sorting data frames

With `arrange` we get to decide which column to sort by. To see the states by murder rate, from lowest to highest, we arrange by `rate` instead:

```
murders %>%
  arrange(rate) %>%
  head()
```

```
##           state abb        region population total       rat
## 1       Vermont  VT     Northeast     625741     2 0.319621
## 2 New Hampshire  NH     Northeast    1316470     5 0.379803
## 3        Hawaii  HI          West    1360301     7 0.514592
## 4  North Dakota  ND North Central     672591     4 0.594715
## 5          Iowa  IA North Central    3046355    21 0.689348
## 6         Idaho  ID          West    1567582    12 0.765510
##   population_in_millions
## 1               0.625741
## 2               1.316470
```

Sorting data frames

Note that the default behavior is to order in ascending order. In **dplyr**, the function `desc` transforms a vector so that it is in descending order. To sort the table in descending order, we can type:

```
murders %>%
  arrange(desc(rate))
```

```
##                    state abb        region population total
## 1   District of Columbia  DC         South     601723    99
## 2              Louisiana  LA         South    4533372   351
## 3               Missouri  MO North Central    5988927   321
## 4               Maryland  MD         South    5773552   293
## 5         South Carolina  SC         South    4625364   207
## 6               Delaware  DE         South     897934    38
## 7               Michigan  MI North Central    9883640   413
## 8            Mississippi  MS         South    2967297   120
## 9                Georgia  GA         South    9920000   376
```

Nested sorting

If we are ordering by a column with ties, we can use a second column to break the tie.Here we order by `region`, then within region we order by murder rate:

```
murders %>%
  arrange(region, rate) %>%
  head()
```

```
##              state abb    region population total      rate po
## 1          Vermont  VT Northeast     625741     2 0.3196211
## 2    New Hampshire  NH Northeast    1316470     5 0.3798036
## 3            Maine  ME Northeast    1328361    11 0.8280881
## 4     Rhode Island  RI Northeast    1052567    16 1.5200933
## 5    Massachusetts  MA Northeast    6547629   118 1.8021791
## 6         New York  NY Northeast   19378102   517 2.6679599
```

The top *n*

If we want to see a larger proportion, we can use the `top_n` function. Here is an example of how to see the top 5 rows:

```
murders %>% top_n(5, rate)
```

```
##                  state abb        region population total
## 1 District of Columbia  DC         South     601723    99 1
## 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
## 1               0.601723
## 2               4.533372
## 3               5.773552
## 4               5.988927
## 5               4.625364
```

Exercises

For these exercises, we will be using the data from the survey collected by the United States National Center for Health Statistics (NCHS). Part of the data is made available via the **NHANES** package. Once you install the **NHANES** package, you can load the data like this:

```
library(NHANES)
data(NHANES)
```

Exercises

The **NHANES** data has many missing values. The `mean` and `sd` functions in R will return `NA` if any of the entries of the input vector is an `NA`. Here is an example:

```
library(dslabs)
data(na_example)
mean(na_example)
```

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

To ignore the NAs we can use the `na.rm` argument:

```
mean(na_example, na.rm = TRUE)
```

```
## [1] 2.301754
```

Exercises

Let's now explore the NHANES data.

1. We will provide some basic facts about blood pressure. First let's select a group to set the standard. We will use 20-to-29-year-old females.

`AgeDecade` is a categorical variable with these ages. Note that the category is coded like " 20-29", with a space in front!

What is the average and standard deviation of systolic blood pressure as saved in the `BPSysAve` variable? Save it to a variable called `ref`.

Hint: Use `filter` and `summarize` and use the `na.rm = TRUE` argument when computing the average and standard deviation. You can also filter the NA values using `filter`.

Exercises

2. Using a pipe, assign the average to a numeric variable `ref_avg`. Hint: Use the code similar to above and then `pull`.

3. Now report the min and max values for the same group.

4. Compute the average and standard deviation for females, but for each age group separately rather than a selected decade as in question 1. Hint: rather than filtering by age and gender, filter by `Gender` and then use `group_by`.

Exercises

5. Repeat exercise 4 for males.

6. We can actually combine both summaries for exercises 4 and 5 into one line of code. This is because `group_by` permits us to group by more than one variable. Obtain one big summary table using `group_by(AgeDecade, Gender)`.

7. For males between the ages of 40-49, compare systolic blood pressure across race as reported in the `Race1` variable. Order the resulting table from lowest to highest average systolic blood pressure.

Tibbles

Tidy data must be stored in data frames. But where is the group information stored in the data frame?

```
murders %>% group_by(region)
```

```
## # A tibble: 51 x 7
## # Groups:   region [4]
##    state                abb   region    population total  rate
##    <chr>                <chr> <fct>          <dbl> <dbl> <dbl>
##  1 Alabama              AL    South        4779736   135  2.82
##  2 Alaska               AK    West          710231    19  2.68
##  3 Arizona              AZ    West         6392017   232  3.63
##  4 Arkansas             AR    South        2915918    93  3.19
##  5 California           CA    West        37253956  1257  3.37
##  6 Colorado             CO    West         5029196    65  1.29
##  7 Connecticut          CT    Northeast    3574097    97  2.71
```

Tibbles

Tibbles are very similar to data frames. In fact, you can think of them as a modern version of data frames.

```
murders %>% group_by(region) %>% class()
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

Tibbles display better

The print method for tibbles is more readable than that of a data frame. To see this, compare the outputs of typing `murders` and the output of murders if we convert it to a tibble.

```
murders

as_tibble(murders)
```

Subsets of tibbles are tibbles

If you subset the columns of a data frame, you may get back an object that is not a data frame, such as a vector or scalar. For example:

```
class(murders[,4])
```

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

is not a data frame. With tibbles this does not happen:

```
class(as_tibble(murders)[,4])
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

This is useful in the tidyverse since functions require data frames as input.

Tibbles can have complex entries

While data frame columns need to be vectors of numbers, strings, or logical values, tibbles can have more complex objects, such as lists or functions. Also, we can create tibbles with functions:

```
tibble(id = c(1, 2, 3), func = c(mean, median, sd))
```

```
## # A tibble: 3 x 2
##      id func
##   <dbl> <list>
## 1     1 <fn>
## 2     2 <fn>
## 3     3 <fn>
```

Tibbles can be grouped

The function `group_by` returns a special kind of tibble: a grouped tibble. This class stores information that lets you know which rows are in which groups. The tidyverse functions, in particular the `summarize` function, are aware of the group information.

Create a tibble using tibble instead of `data.frame`

To create a data frame in the tibble format, you can do this by using the `tibble` function.

```
grades <- tibble(names = c("John", "Juan", "Jean", "Yao"),
                 exam_1 = c(95, 80, 90, 85),
                 exam_2 = c(90, 85, 85, 90))
```

Create a tibble using tibble instead of data.frame

Note that base R (without packages loaded) has a function with a very similar name, `data.frame`, that can be used to create a regular data frame rather than a tibble.

```
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),
                     exam_1 = c(95, 80, 90, 85),
                     exam_2 = c(90, 85, 85, 90))
```

To convert a regular data frame to a tibble, you can use the `as_tibble` function.

```
as_tibble(grades) %>% class()
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

The dot operator

One of the advantages of using the pipe %>% is that we do not have to keep naming new objects as we manipulate the data frame. If we want to compute the median murder rate for states in the southern states, instead of typing:

```
tab_1 <- filter(murders, region == "South")
tab_2 <- mutate(tab_1, rate = total / population * 10^5)
rates <- tab_2$rate
median(rates)
```

```
## [1] 3.398069
```

The dot operator

We can avoid defining any new intermediate objects by instead typing:

```
filter(murders, region == "South") %>%
  mutate(rate = total / population * 10^5) %>%
  summarize(median = median(rate)) %>%
  pull(median)
```

```
## [1] 3.398069
```

The dot operator

What if the pull function was not available and we wanted to access tab_2$rate? What data frame name would we use? The answer is the dot operator.

```
rates <-   filter(murders, region == "South") %>%
  mutate(rate = total / population * 10^5) %>%
  .$rate
median(rates)
```

```
## [1] 3.398069
```

The purrr package

We learned about the `sapply` function. We constructed a function and used `sapply` to compute the sum of the first `n` integers for several values of `n` like this:

```
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
n <- 1:25
s_n <- sapply(n, compute_s_n)
```

The purrr package

The **purrr** package includes functions similar to `sapply` but that better interact with other tidyverse functions. The first **purrr** function we will learn is `map`, which works very similar to `sapply` but always, without exception, returns a list:

```
library(purrr)
s_n <- map(n, compute_s_n)
class(s_n)
```

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

The purrr package

If we want a numeric vector, we can instead use map_dbl which always returns a vector of numeric values.

```
s_n <- map_dbl(n, compute_s_n)
class(s_n)
```

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

This produces the same results as the `sapply` call shown above.

The purrr package

A particularly useful **purrr** function for interacting with the rest of the tidyverse is `map_df`, which always returns a tibble data frame.

However, the function being called needs to return a vector or a list with names. For this reason, the following code would result in a `Argument 1 must have names` error:

```
s_n <- map_df(n, compute_s_n)
# Error: Argument 1 must have names.
```

The purrr package

We need to change the function to make this work:

```
compute_s_n <- function(n){
  x <- 1:n
  tibble(sum = sum(x))
}
s_n <- map_df(n, compute_s_n)
```

The **purrr** package provides much more functionality not covered here. For more details you can consult this online resource.

Tidyverse conditionals

A typical data analysis will often involve one or more conditional operations. In this section we present two **dplyr** functions that provide further functionality for performing conditional operations.

case_when

The case_when function is useful for vectorizing conditional statements. It is similar to ifelse but can output any number of values, as opposed to just TRUE or FALSE. Here is an example splitting numbers into negative, positive, and 0:

```
x <- c(-2, -1, 0, 1, 2)
case_when(x < 0 ~ "Negative",
          x > 0 ~ "Positive",
          TRUE  ~ "Zero")
```

```
## [1] "Negative" "Negative" "Zero"     "Positive" "Positive"
```

A common use for this function is to define categorical variables based on existing variables.

case_when

For example, suppose we want to compare the murder rates in four groups of states: *New England*, *West Coast*, *South*, and *other*. Here is how we use case_when to do this:

```
murders %>%
  mutate(group = case_when(
    abb %in% c("ME", "NH", "VT", "MA", "RI", "CT") ~ "New Engl
    abb %in% c("WA", "OR", "CA") ~ "West Coast",
    region == "South" ~ "South",
    TRUE ~ "Other")) %>%
  group_by(group) %>%
  summarize(rate = sum(total) / sum(population) * 10^5)
```

`between`

A common operation in data analysis is to determine if a value falls inside an interval. For example, to check if the elements of a vector `x` are between `a` and `b` we can type

```
x >= a & x <= b
```

However, this can become cumbersome, especially within the tidyverse approach. The `between` function performs the same operation.

```
between(x, a, b)
```

Exercises

1. Load the `murders` dataset. Which of the following is true?

a. `murders` is in tidy format and is stored in a tibble.
b. `murders` is in tidy format and is stored in a data frame.
c. `murders` is not in tidy format and is stored in a tibble.
d. `murders` is not in tidy format and is stored in a data frame.

2. Use `as_tibble` to convert the `murders` data table into a tibble and save it in an object called `murders_tibble`.

Exercises

3. Use the `group_by` function to convert `murders` into a tibble that is grouped by region.

4. Write tidyverse code that is equivalent to this code:

```
exp(mean(log(murders$population)))
```

Write it using the pipe so that each function is called without arguments. Use the dot operator to access the population. Hint: The code should start with `murders %>%`.

Exercises

5. Use the `map_df` to create a data frame with three columns named `n`, `s_n`, and `s_n_2`. The first column should contain the numbers 1 through 100. The second and third columns should each contain the sum of 1 through n with n the row number.