Loops in R

Introduction

At the heart of programming is the concept of repeating a task multiple times. A `for` loop is one fundamental way to do that. Loops enable efficient repetition, saving time and effort.

Mastering this concept is essential for writing intelligent and efficient R code.

Let's dive in and enhance your coding skills!

Learning Objectives

By the end of this lesson, you will be able to:

- Explain the syntax and structure of a basic `for` loop in R
- Use index variables to iterate through multiple vectors simultaneously in a loop
- Integrate `if/else` conditional statements within a loop
- Store loop results in vectors and lists
- Apply loops to tasks like analyzing multiple datasets and generating multiple plots

Packages

This lesson will require the following packages to be installed and loaded:

```
# Load packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, here, openxlsx, tools, outbreaks, medicaldata)
```

Intro to `for` Loops

Let's start with a simple example. Suppose we have a vector of children's ages in years, and we want to convert these to months:

```
ages <- c(7, 8, 9) # Vector of ages in years
```

We can do this easily with the * operation in R:

```
ages * 12
```

```
## [1]  84  96 108
```

But let's walk through how we could accomplish this using a `for` loop instead, since that is (conceptually) what R is doing under the hood.

```
for (age in ages) print(age * 12)
```

```
## [1] 84
## [1] 96
## [1] 108
```

In this loop, `age` is a temporary variable that takes the value of each element in `ages` during each iteration. First, `age` is 7, then 8, then 9.

You can choose any name for this variable:

```
for (random_name in ages) print(random_name * 12)
```

```
## [1] 84
## [1] 96
## [1] 108
```

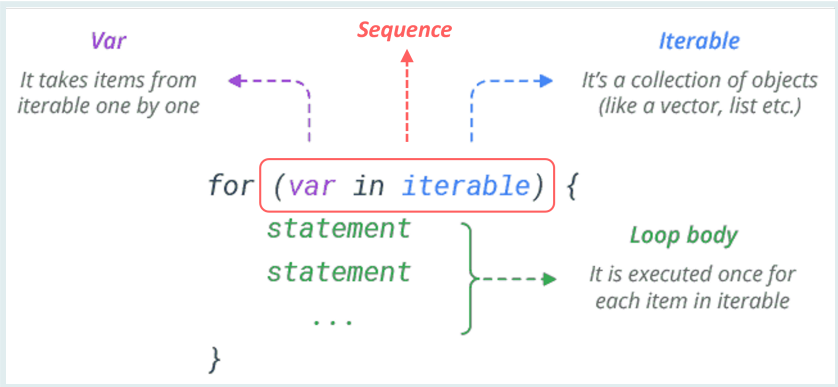
If the content of the loop is more than one line, you need to use curly brackets `{}` to indicate the body of the loop.

```
for (age in ages) {
  month_age = age * 12
  print(month_age)
}
```

```
## [1] 84
## [1] 96
```

```
## [1] 108
```

The general structure of any `for` loop is illustrated in the diagram below:



(NOTE: Answers are at the bottom of the page. Try to answer the questions yourself before checking.)

PRACTICE (in RMD)

Hours to Minutes Basic Loop

Try converting hours to minutes using a `for` loop. Start with this vector of hours:

```
hours <- c(3, 4, 5) # Vector of hours
# Your code here

for ___
_____ # convert hours to minutes and print
```

SIDE NOTE

Loops can be nested within each other. For instance:

```
for (i in 1:2) {
  for (j in 1:2) {
    print(i * j)
  }
}
```

```
## [1] 1
## [1] 2
## [1] 2
## [1] 4
```

SIDE NOTE

This creates a combination of `i` and `j` values as shown in this table:

i	j	i * j
1	1	1
1	2	2
2	1	2
2	2	4

Nested loops are less common though, and often have more efficient alternatives.

Are `for` Loops Useful in R?

While `for` loops are foundational in many programming languages, their usage in R is somewhat less frequent. This is because R inherently handles *vectorized* operations, automatically applying a function to each element of a vector.

For example, our initial age conversion could be achieved without a loop:

```
ages * 12
```

```
## [1]  84  96 108
```

Moreover, R typically deals with data frames rather than raw vectors. For data frames, we often use functions from the `tidyverse` package to apply operations across columns:

```
ages_df <- tibble(age = ages)
ages_df %>%
  mutate(age_months = age * 12)
```

```
## # A tibble: 3 × 2
##     age age_months
##   <dbl>      <dbl>
## 1     7         84
## 2     8         96
## 3     9        108
```

However, there are scenarios where loops are useful, especially when working with multiple data frames or non-dataframe (sometimes called *non-rectangular*) objects.

We will explore these later in the lesson, but first we'll spend some more time getting comfortable with loops using toy examples.

Loops vs function mapping

It's important to note that loops can often be replaced by custom functions which are then mapped across a vector or data frame.

We're teaching loops nonetheless because they are quite easy to learn, reason about and debug, even for beginners.

Looping with an Index

It is often useful to loop through a vector using an index (plural: indices), which is a counter that keeps track of the current iteration.

Let's look at our `ages` vector again, which we want to convert to months:

```
ages <- c(7, 8, 9) # Vector of ages in years
```

To use indices in a loop, we first create a sequence that represents each position in the vector:

```
1:length(ages) # Create a sequence of indices that is the same length as ages
```

```
## [1] 1 2 3
```

```
indices <- 1:length(ages)
```

Now, `indices` has values 1, 2, 3, corresponding to the positions in `ages`. We use this in a `for` loop as follows:

```
for (i in indices) {
  print(ages[i] * 12)
}
```

```
## [1] 84
## [1] 96
## [1] 108
```

In this code, `ages[i]` refers to the `i`th element in our `ages` list.

The name of the variable `i` is arbitrary. We could have used `j` or `index` or `position` or anything else.

```
for (position in indices) {
  print(ages[position] * 12)
}
```

```
## [1] 84
## [1] 96
## [1] 108
```

Often we do not need to create a separate variable for the indices. We can just use the `:` operator to create a sequence directly in the `for` loop:

```
for (i in 1:length(ages)) {
  print(ages[i] * 12)
}
```

```
## [1] 84
## [1] 96
## [1] 108
```

Such index-based loops are useful for working with multiple vectors at the same time. We will see this in the next section.

Hours to Minutes Indexed Loop

Rewrite your loop from last question using indices:

```
hours <- c(3, 4, 5) # Vector of hours

# Your code here

for ___ {
  ___
}
```

SIDE NOTE

The function `seq_along()` is a shortcut for creating a sequence of indices. It is equivalent to `1:length()`:

```
# These two are equivalent:
seq_along(ages)
```

```
## [1] 1 2 3
```

```
1:length(ages)
```

```
## [1] 1 2 3
```

Looping on Multiple Vectors

Looping with indices allows us to work with multiple vectors simultaneously. Suppose we have vectors for ages and heights:

```
ages <- c(7, 8, 9) # ages in years
heights <- c(120, 130, 140) # heights in cm
```

We can loop through both using the index method:

```
for(i in 1:length(ages)) {
  age <- ages[i]
  height <- heights[i]

  print(paste("Age:", age, "Height:", height))
}
```

```
## [1] "Age: 7 Height: 120"
## [1] "Age: 8 Height: 130"
## [1] "Age: 9 Height: 140"
```

In each iteration: - `i` is the index. - We extract the ith element from each vector

and print it.

Alternatively, we can skip the variable assignment and use the indices in the `print()` statement directly:

```
for(i in 1:length(ages)) {
  print(paste("Age:", ages[i], "Height:", heights[i]))
}
```

```
## [1] "Age: 7 Height: 120"
## [1] "Age: 8 Height: 130"
## [1] "Age: 9 Height: 140"
```

BMI Calculation Loop

Using a for loop, calculate the Body Mass Index (BMI) of the three individuals shown below. The formul for BMI is `BMI = weight / (height ^ 2)`.


```
weights <- c(30, 32, 35) # Weights in kg
heights <- c(1.2, 1.3, 1.4) # Heights in meters

for(i in ____________________) {

  ____________________________

  print(paste("Weight:", ____,
              "Height:", ____,
              "BMI:", ____,
              ))

}
```

Storing Loop Results

In most cases, you'll want to store the results of a loop rather than just printing them as we have been doing above. Let's look at how to do this.

Consider our age-to-months example:

```r
ages <- c(7, 8, 9)

for (age in ages) {
  print(paste(age * 12, "months"))
}
```

```
## [1] "84 months"
## [1] "96 months"
## [1] "108 months"
```

To store these converted ages, we first create an empty vector:

```r
ages_months <- vector(mode = "numeric", length = length(ages))
# This can also be written as:
ages_months <- vector("numeric", length(ages))

ages_months # Shows the empty vector
```

```
## [1] 0 0 0
```

This creates a numeric vector of the same length as `ages`, initially filled with zeros. To store a value in the vector, we do the following:

```r
ages_months[1] <- 99 # Store 99 in the first element of ages_months
ages_months[2] <- 100 # Store 100 in the second element of ages_months
ages_months
```

```
## [1]  99 100   0
```

Now, let's execute the loop, storing the results in `ages_months`:

```r
ages_months <- vector("numeric", length(ages))

for (i in 1:length(ages)) {
  ages_months[i] <- ages[i] * 12
}
ages_months
```

```
## [1]  84  96 108
```

In this loop:

- On the first iteration, `i` is 1. We multiply the first element of `ages` by 12 and store it in the first element of `ages_months`.

- Then `i` is 2, then 3. In each iteration, we multiply the corresponding element of `ages` by 12 and store it in the corresponding element of `ages_months`.

Height cm to m

Use a for loop to convert height measurements from cm to m. Store the results in a vector called `height_meters`.

```
height_cm <- c(180, 170, 190, 160, 150) # Heights in cm

height_m <- vector(_______________-) # numeric vector of same
length as heigh_cm

for ___ {
  height_m[i] <- ________________________
}
```

In order to save the results from your iteration, you must create your empty object **outside** the loop. Otherwise, you will only save the result of the last iteration.

This is a common mistake. Consider the below as an example:

```
ages <- c(7, 8, 9)

for (i in 1:length(ages)) {
  ages_months <- vector("numeric", length(ages))
  ages_months[i] <- ages[i] * 12
}
ages_months
```

```
## [1]   0   0 108
```

Do you see the problem?

If you are in a rush, you can skip using the `vector()` function and initialize your vector with `c()` instead, then progressively fill it with values by index:

```r
ages_months <- c()

for (i in 1:length(ages)) {
  ages_months[i] <- ages[i] * 12
}
ages_months
```

```
## [1]  84  96 108
```

SIDE NOTE

And you can also skip the index and use `c()` to append values to the end of the vector:

```r
ages_months <- c()

for (age in ages) {
  ages_months <- c(ages_months, age * 12)
}
ages_months
```

```
## [1]  84  96 108
```

However, in both of these cases, R does not know the final length of the vector as it's going through the iterations, so it has to reallocate memory at each iteration. This can cause slow performance if you are working with large vectors.

If Statements in Loops

Just as `if` statements can be used in functions, they can be integrated into loops.

Consider this example:

```r
age_vec <- c(2, 12, 17, 24, 60) # Vector of ages

for (age in age_vec) {
  if (age < 18) print(paste("Child, Age", age ))
}
```

```
## [1] "Child, Age 2"
## [1] "Child, Age 12"
```

```
## [1] "Child, Age 17"
```

It is often clearer to use curly braces to indicate the `if` statement's body. It also allows us to add more lines of code to the body of the `if` statement:

```
for (age in age_vec) {
  if (age < 18) {
    print("Processing:")
    print(paste("Child, Age", age ))
  }
}
```

```
## [1] "Processing:"
## [1] "Child, Age 2"
## [1] "Processing:"
## [1] "Child, Age 12"
## [1] "Processing:"
## [1] "Child, Age 17"
```

Let's add another condition to classify as 'Child' or 'Teen':

```
for (age in age_vec) {
  if (age < 13) {
    print(paste("Child, Age", age))
  } else if (age >= 13 && age < 18) {
    print(paste("Teen, Age", age))
  }
}
```

```
## [1] "Child, Age 2"
## [1] "Child, Age 12"
## [1] "Teen, Age 17"
```

We can include a single `else` statement at the end to catch all other ages:

```
for (age in age_vec) {
  if (age < 13) {
    print(paste("Child, Age", age))
  } else if (age >= 13 && age < 18) {
    print(paste("Teen, Age", age))
  } else {
    print(paste("Adult, Age", age))
  }
}
```

```
## [1] "Child, Age 2"
## [1] "Child, Age 12"
## [1] "Teen, Age 17"
```

```
## [1] "Adult, Age 24"
## [1] "Adult, Age 60"
```

To store these classifications, we can create an empty vector, and use an index-based loop to store the results:

```r
age_class <- vector("character", length(age_vec)) # Create empty vector
for (i in 1:length(age_vec)) {
  if (age_vec[i] < 13) {
    age_class[i] <- "Child"
  } else if (age_vec[i] >= 13 && age_vec[i] < 18) {
    age_class[i] <- "Teen"
  } else {
    age_class[i] <- "Adult"
  }
}
age_class
```

```
## [1] "Child" "Child" "Teen"  "Adult" "Adult"
```

Temperature Classification

You have a vector of body temperatures in Celsius. Classify each temperature as 'Hypothermia', 'Normal', or 'Fever' using a `for` loop combined with `if` and `else` statements.

Use these rules:

- Below 36.0°C: 'Hypothermia'
- Between 36.0°C and 37.5°C: 'Normal'
- Above 37.5°C: 'Fever'


```
body_temps <- c(35, 36.5, 37, 38, 39.5) # Body temperatures in
Celsius
classif_vec <- vector(_____________________) # character vec,
length of body_temps
for (i in 1:length(________)) {
    # Add your if-else logic here
  if (body_temps[i] < 36.0) {
    out <- "Hypothermia"
  } ## add other conditions


    # Final print statement
    classif_vec[i] <- paste(body_temps[i], "°C is", out)
}
classif_vec
```

An expected output is below

```
35°C is Hypothermia
36.5°C is Normal
37°C is Normal
38°C is Fever
39.5°C is Fever
```

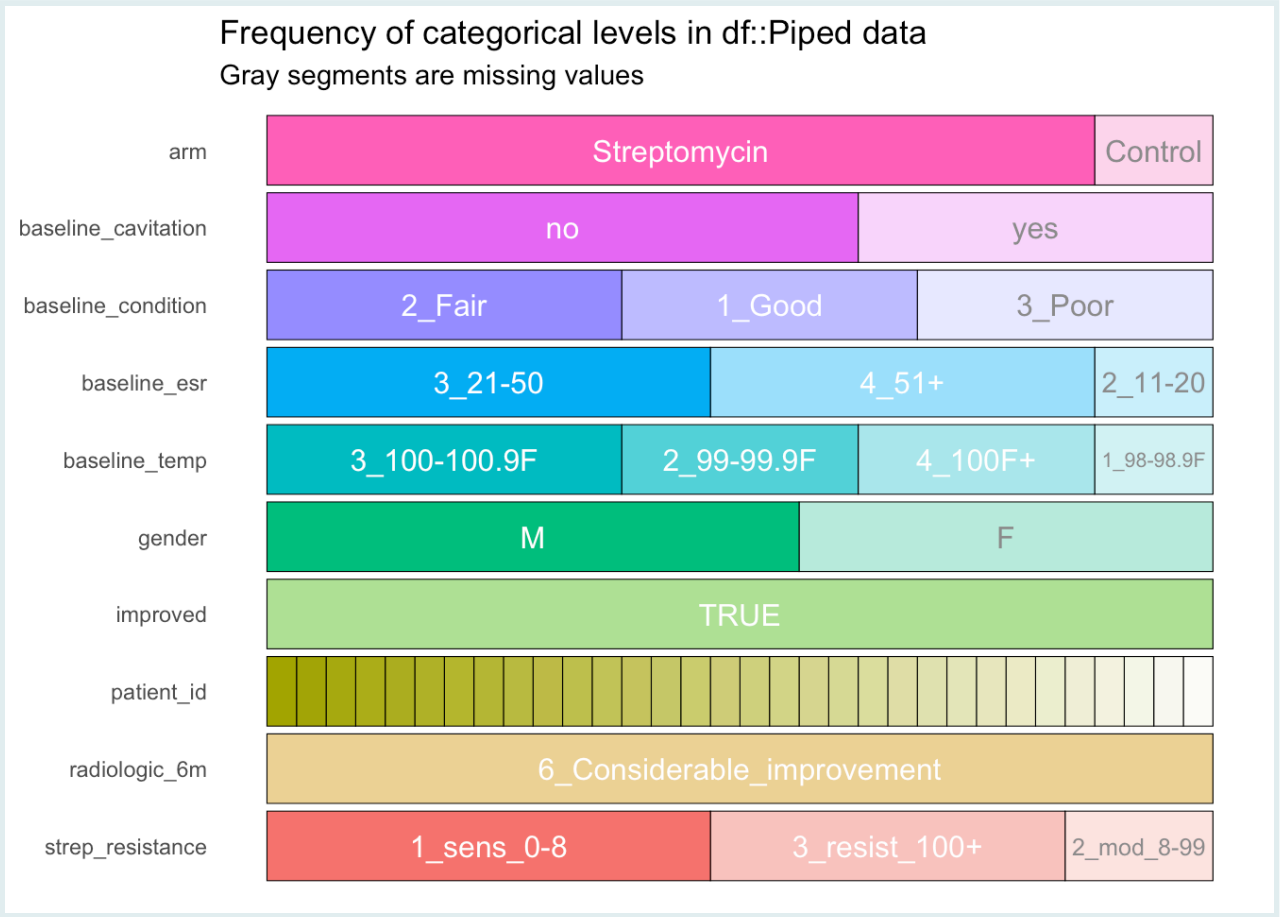
Real Loops Application: Generating Multiple Plots

Now that you have a solid understanding of `for` loops, let's apply our knowledge to a more realistic looping task: generating multiple plots.

We'll use the `strep_tb` dataset from the `medicaldata` package to demonstrate this. Our aim is to create category inspection plots for each radiologic 6-month improvement group.

Let's start by creating a plot for one of the groups. We'll use `inspectdf::inspect_cat()` to generate a category inspection plot:

```
cat_plot <-
  medicaldata::strep_tb %>%
  filter(radiologic_6m == "6_Considerable_improvement") %>%
  inspectdf::inspect_cat() %>%
  inspectdf::show_plot()
cat_plot
```



This plot gives us a quick way to visualize the distribution of categories in our dataset.

Now, we want to create similar plots for each radiologic improvement group in the dataset. First, let's identify all the unique groups using the `unique` function:

```
radiologic_levels_6m <- medicaldata::strep_tb$radiologic_6m %>% unique()
radiologic_levels_6m
```

```
## [1] 6_Considerable_improvement   5_Moderate_improvement       4_No_change
## [4] 3_Moderate_deterioration     2_Considerable_deterioration 1_Death
## 6 Levels: 6_Considerable_improvement 5_Moderate_improvement ... 1_Death
```

Next, we'll initiate an empty list object where we will store the plots.

```
cat_plot_list <- vector("list", length(radiologic_levels_6m))
cat_plot_list
```

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

```
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
##
## [[6]]
## NULL
```

We will also set the names of the list elements to the radiologic improvement groups. This is an optional step, but it makes it easier to access specific plots later on.

```
names(cat_plot_list) <- radiologic_levels_6m
cat_plot_list
```

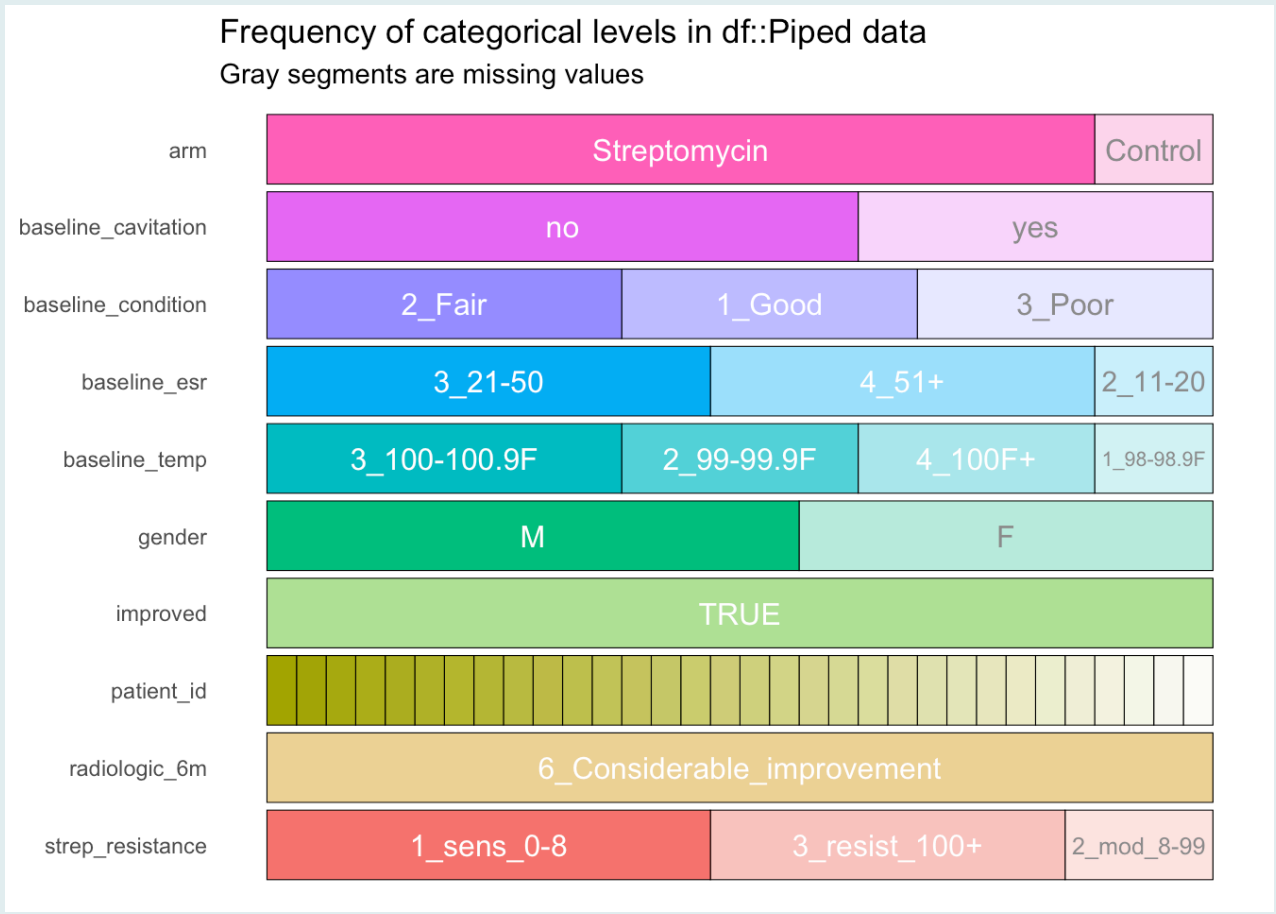
```
## $`6_Considerable_improvement`
## NULL
##
## $`5_Moderate_improvement`
## NULL
##
## $`4_No_change`
## NULL
##
## $`3_Moderate_deterioration`
## NULL
##
## $`2_Considerable_deterioration`
## NULL
##
## $`1_Death`
## NULL
```

Finally, we'll use a loop to generate a plot for each group and store it in the list:

```
for (level in radiologic_levels_6m) {

  # Generate plot for each level
  cat_plot <-
    medicaldata::strep_tb %>%
    filter(radiologic_6m == level) %>%
    inspectdf::inspect_cat() %>%
    inspectdf::show_plot()

  # Append to the list
  cat_plot_list[[level]] <- cat_plot
}
```

To access a specific plot, we can use the double bracket syntax:

```
cat_plot_list[["6_Considerable_improvement"]]
```



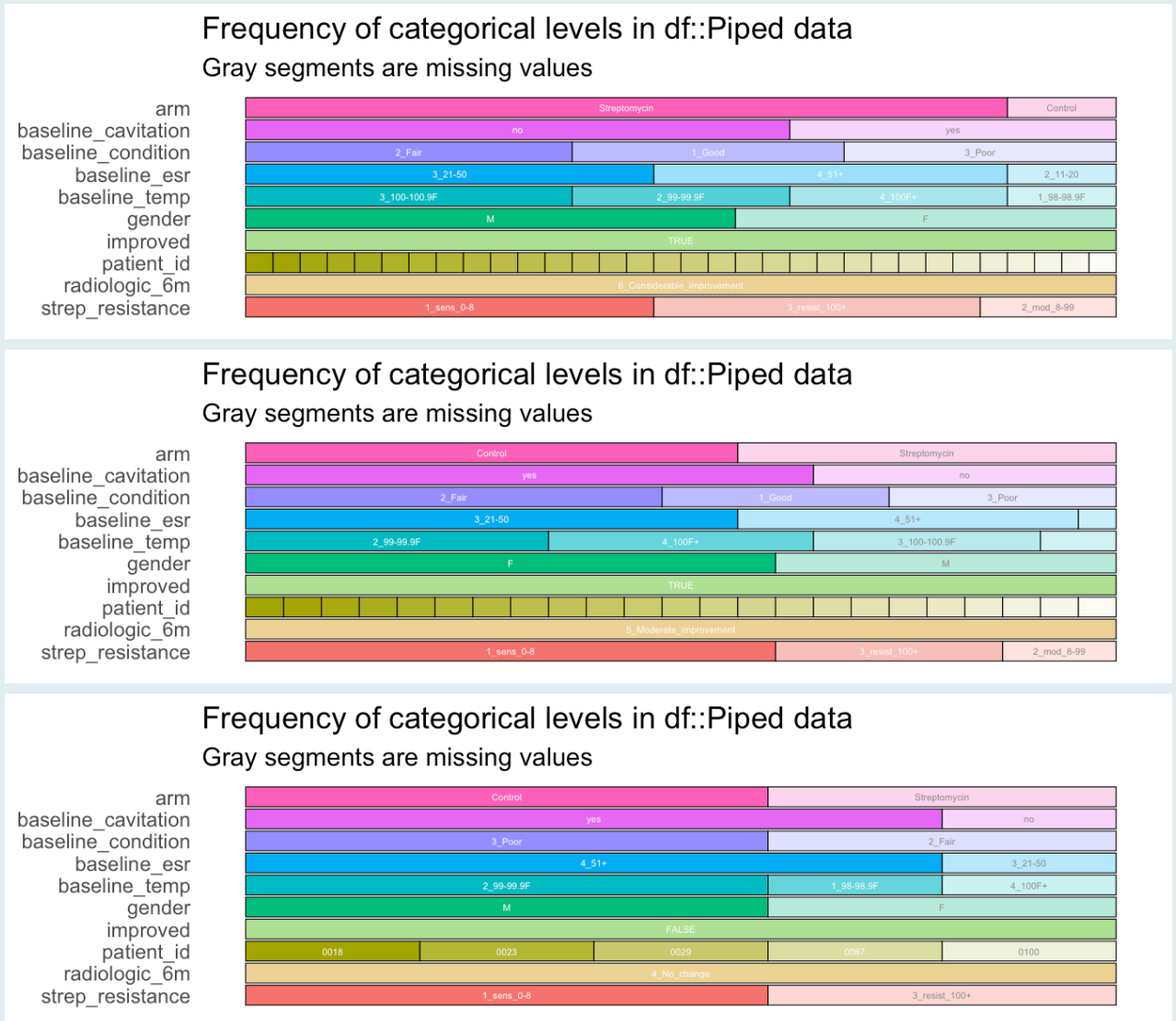
Note that in this case, the list elements are *named*, rather than just numbered. This is because we used the `level` variable as the index in the loop.

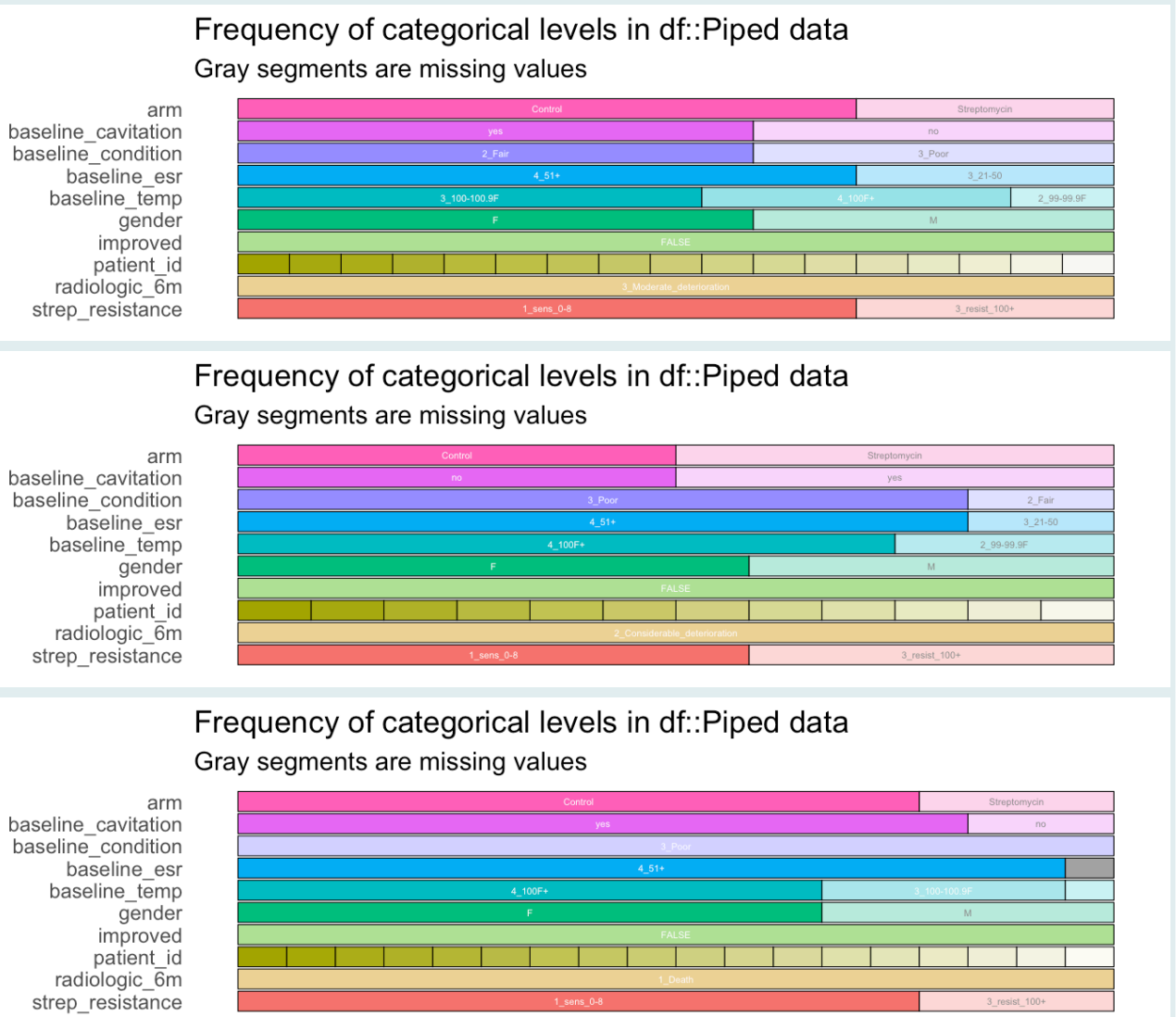
To display all plots at once, we simply call the entire list.

```
cat_plot_list
```

```
## $`6_Considerable_improvement`


##
## $`5_Moderate_improvement`


##
## $`4_No_change`
```

```
##
## $`3_Moderate_deterioration`


##
## $`2_Considerable_deterioration`


##
## $`1_Death`
```





Visualizing TB Cases

In this exercise, you will use WHO data from the `tidyr` package to create line graphs showing the number of new TB cases in children over the years in South American countries.

First, we'll prepare the data:

```
tb_child_cases <- tidyr::who2 %>%
  transmute(country, year,
            tb_cases_children = sp_m_014 + sp_f_014 + sn_m_014
+ sn_f_014) %>%
  filter(country %in% c("Brazil", "Colombia", "Argentina",
```

```
## Error: 'who2' is not an exported object from
'namespace:tidyr'
```

```
tb_child_cases
```

```
## Error in eval(expr, envir, enclos): object 'tb_child_cases'
not found
```

Now, fill in the blanks in the template below to create a line graph for each country using a `for` loop:


```
# Get list of countries. Hint: Use unique() on the country
column
countries <- ____________________________________________

# Create list to store plots. Hint: Initialize an empty list
tb_child_cases_plots <- vector("list", _________________)
names(tb_child_cases_plots) <- countries # Set names of list
elements

# Loop through countries
for (country in _____________) {

  # Filter data for each country
  tb_child_cases_filtered <-
_____________________________________________

  # Make plot
  tb_child_cases_plot <-
_____________________________________________

  # Append to list. Hint: Use double brackets
  tb_child_cases_plots[[country]] <- tb_child_cases_plot
}

tb_child_cases_plots
```

```
## Error: <text>:2:15: unexpected input
## 1: # Get list of countries. Hint: Use unique() on the
country column
## 2: countries <- __
##                  ^
```

Wrap Up!

In this lesson, we delved into for loops in R, demonstrating their utility from basic tasks to complex data analysis involving multiple datasets and plot generation. Despite R's preference for vectorized operations, for loops are indispensable in certain scenarios. Hopefully, this lesson has equipped you with the skills to confidently implement for loops in various data processing contexts.

Answer Key

Hours to Minutes Basic Loop

```
hours <- c(3, 4, 5) # Vector of hours

for (hour in hours) {
  minutes <- hour * 60
  print(minutes)
}
```

```
## [1] 180
## [1] 240
## [1] 300
```

Hours to Minutes Indexed Loop

```
hours <- c(3, 4, 5) # Vector of hours

for (i in 1:length(hours)) {
  minutes <- hours[i] * 60
  print(minutes)
}
```

```
## [1] 180
## [1] 240
## [1] 300
```

BMI Calculation Loop

```
weights <- c(30, 32, 35) # Weights in kg
heights <- c(1.2, 1.3, 1.4) # Heights in meters

for(i in 1:length(weights)) {
  bmi <- weights[i] / (heights[i] ^ 2)

  print(paste("Weight:", weights[i],
              "Height:", heights[i],
              "BMI:", bmi))
}
```

```
## [1] "Weight: 30 Height: 1.2 BMI: 20.8333333333333"
## [1] "Weight: 32 Height: 1.3 BMI: 18.9349112426035"
## [1] "Weight: 35 Height: 1.4 BMI: 17.8571428571429"
```

Height cm to m

```
height_cm <- c(180, 170, 190, 160, 150) # Heights in cm

height_m <- vector("numeric", length = length(height_cm))

for (i in 1:length(height_cm)) {
  height_m[i] <- height_cm[i] / 100
}
height_m
```

```
## [1] 1.8 1.7 1.9 1.6 1.5
```

Temperature Classification

```
body_temps <- c(35, 36.5, 37, 38, 39.5) # Body temperatures in Celsius
classif_vec <- vector("character", length = length(body_temps)) # character
vector

for (i in 1:length(body_temps)) {
    # Add your if-else logic here
    if (body_temps[i] < 36) {
        out <- "Hypothermia"
    } else if (body_temps[i] <= 37.5) {
        out <- "Normal"
    } else {
        out <- "Fever"
    }

    # Final print statement
    classif_vec[i] <- paste(body_temps[i], "°C is", out)
}
classif_vec
```

```
## [1] "35 °C is Hypothermia" "36.5 °C is Normal"    "37 °C is Normal"
## [4] "38 °C is Fever"       "39.5 °C is Fever"
```

Visualizing TB Cases

```
# Assuming tb_child_cases is a dataframe with the necessary columns
countries <- unique(tb_child_cases$country)
```

```
## Error in unique(tb_child_cases$country): object 'tb_child_cases' not found
```

```
# Create list to store plots
tb_child_cases_plots <- vector("list", length(countries))
```

```
## Error in vector("list", length(countries)): object 'countries' not found
```

```
names(tb_child_cases_plots) <- countries
```

```
## Error in eval(expr, envir, enclos): object 'countries' not found
```

```r
# Loop through countries
for (countryname in countries) {

  # Filter data for each country
  tb_child_cases_filtered <- filter(tb_child_cases, country == countryname)

  # Make plot
  tb_child_cases_plot <- ggplot(tb_child_cases_filtered, aes(x = year, y =
tb_cases_children)) +
    geom_line() +
    ggtitle(paste("TB Cases in Children –", countryname))

  # Append to list
  tb_child_cases_plots[[countryname]] <- tb_child_cases_plot
}
```

```
## Error in eval(expr, envir, enclos): object 'countries' not found
```

```r
tb_child_cases_plots[["Uruguay"]]
```

```
## Error in eval(expr, envir, enclos): object 'tb_child_cases_plots' not
found
```

Contributors

The following team members contributed to this lesson:

SABINA RODRIGUEZ VELÁSQUEZ

Project Manager and Scientific Collaborator, The GRAPH Network
Infectiously enthusiastic about microbes and Global Health

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

Some material in this lesson was adapted from the following sources:

- Barnier, Julien. “Introduction à R et au tidyverse.” https://juba.github.io/tidyverse
- Wickham, Hadley; Grolemund, Garrett. “R for Data Science.” https://r4ds.had.co.nz/
- Wickham, Hadley; Grolemund, Garrett. “R for Data Science (2e).” https://r4ds.hadley.nz/