Types, vectors, and functions in R

2021-01-24

Vectors and Types

Vectors

c(1, 3, 5)

c(TRUE, FALSE, TRUE, TRUE)

c("red", "blue")

Vectors

Vectors

Vectors have 1 dimension

Vectors

Vectors have 1 dimension

Vectors have a length.

```
length(c("blue", "red"))
```

Vectors

Vectors have 1 dimension

Vectors have a length.

```
length(c("blue", "red"))
```

Some vectors have names.

```
names(c("x" = 1, "y = 1))
```

Vectors

Vectors have 1 dimension

Vectors have a length.

length(c("blue", "red"))

Some vectors have names.

names(c("x" = 1, "y = 1))

Vectors have types

Types

Numeric/double

Integer

Factor

Character

Logical

Dates

Packages to work with types:

Strings/character: stringr

Packages to work with types:

Strings/character: stringr

Factors: forcats

Packages to work with types:

Strings/character: stringr

Factors: forcats

Dates: lubridate

Making vectors

```
1:3
```

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

```
c(1, 2, 3)
```

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

```
rep(1, 3)
```

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

```
seq(from = 1, to = 3, by = .5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0
```

Your Turn 1

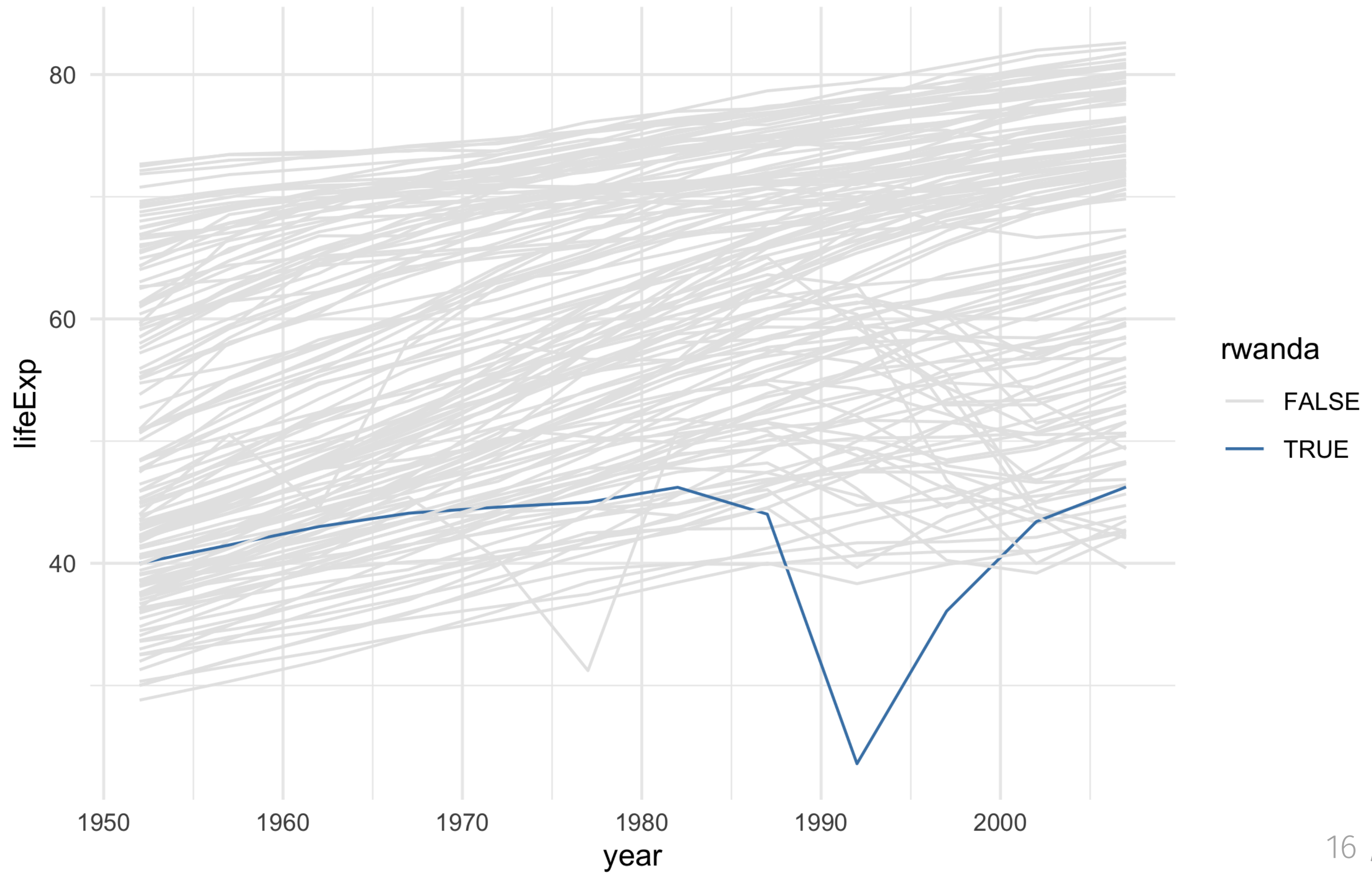
Create a character vector of colors using `c()`**. Use the colors "grey90" and "steelblue". Assign the vector to a name.**

Use the vector you just created to change the colors in the plot below using `scale_color_manual()`**. Pass it using the** `values` **argument.**

Your Turn 1

```r
cols <- c("grey90", "steelblue")

gapminder %>%
  mutate(rwanda = ifelse(country == "Rwanda", TRUE, FALSE)) %>%
  ggplot(aes(year, lifeExp, color = rwanda, group = country)) +
  geom_line() +
  scale_color_manual(values = cols) +
  theme_minimal()
```

Your Turn 1


80
60
40
lifeExp
1950
1960
1970
1980
1990
2000
year
rwanda
FALSE
TRUE

Working with vectors

Subset vectors with [] or [[]]

```r
x <- c(1, 5, 7)
```

```r
x[2]
```

```
## [1] 5
```

```r
x[[2]]
```

```
## [1] 5
```

```r
x[c(FALSE, TRUE, FALSE)]
```

```
## [1] 5
```

Working with vectors

Modify elements

```
x
```

```
## [1] 1 5 7
```

```
x[2] <- 100
```

```
x
```

```
## [1]   1 100   7
```

Modify elements

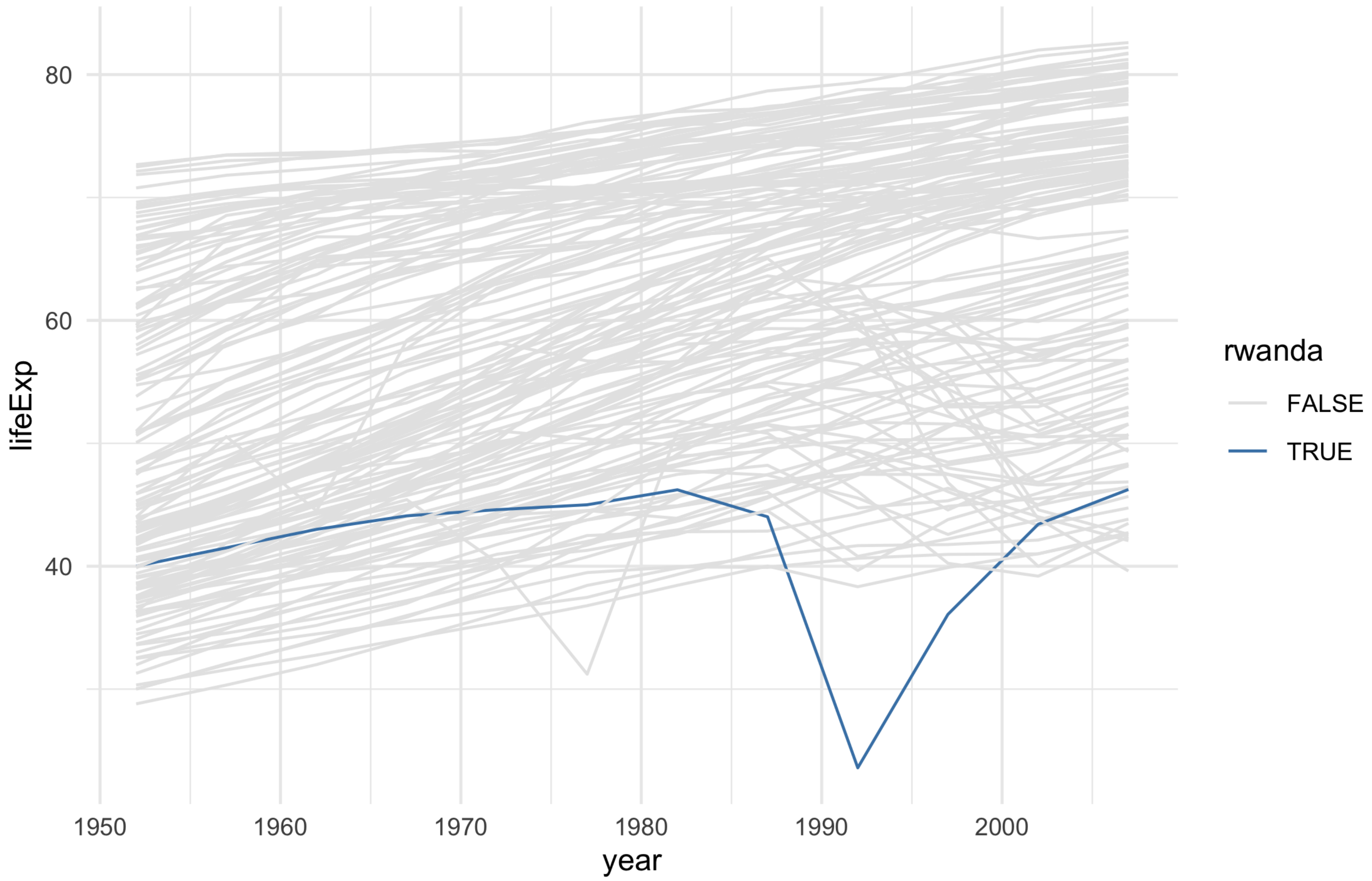
```
x
```

```
## [1]   1 100   7
```

```
x[x > 10] <- NA
```

```
x
```

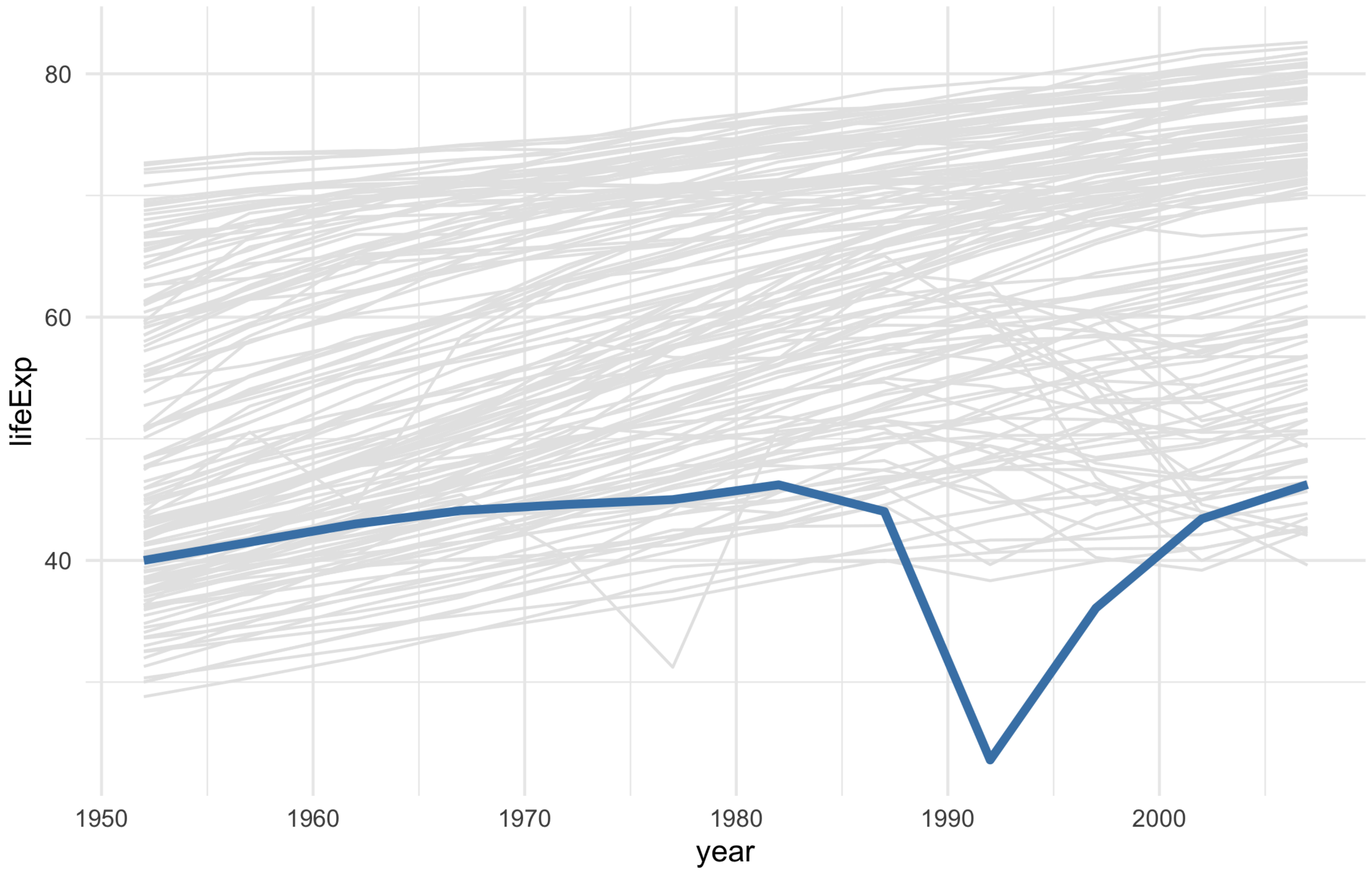
```
## [1]  1 NA  7
```


80
60
40
lifeExp
1950
1960
1970
1980
1990
2000
year
rwanda
FALSE
TRUE

```r
cols <- c("grey90", "steelblue")

gapminder %>%
  mutate(rwanda = ifelse(country == "Rwanda", TRUE, FALSE)) %>%
  ggplot(aes(year, lifeExp, color = rwanda, group = country)) +
  geom_line() +
  scale_color_manual(values = cols) +
  theme_minimal()
```

```r
cols <- c("grey90", "steelblue")

gapminder %>%
  mutate(rwanda = ifelse(country == "Rwanda", TRUE, FALSE)) %>%
  ggplot(aes(year, lifeExp, group = country)) +
  geom_line(
    data = function(x) filter(x, !rwanda),
    color = cols[1]
  ) +
  theme_minimal()
```

```r
cols <- c("grey90", "steelblue")

gapminder %>%
  mutate(rwanda = ifelse(country == "Rwanda", TRUE, FALSE)) %>%
  ggplot(aes(year, lifeExp, color = rwanda, group = country)) +
  geom_line(
    data = function(x) filter(x, !rwanda),
    color = cols[1]
  ) +
  geom_line(
    data = function(x) filter(x, rwanda),
    color = cols[2],
    size = 1.5
  ) +
  theme_minimal()
```



80
60
40
lifeExp
1950
1960
1970
1980
1990
2000
year

Your Turn 2

Create a numeric vector that has the following values: 3, 5, NA, 2, and NA.

Try using `sum()`**. Then add** `na.rm = TRUE`**.**

Check which values are missing with `is.na()`**; save the results to a new object and take a look**

Change all missing values of `x` **to 0**

Try `sum()` **again without** `na.rm = TRUE`**.**

Your Turn 2

```r
x <- c(3, 5, NA, 2, NA)
sum(x)
```

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

Your Turn 2

```r
sum(x, na.rm = TRUE)
```

```
## [1] 10
```

Your Turn 2

```r
x_missing <- is.na(x)
x_missing
```

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

```r
x[x_missing] <- 0
x
```

```
## [1] 3 5 0 2 0
```

```r
sum(x)
```

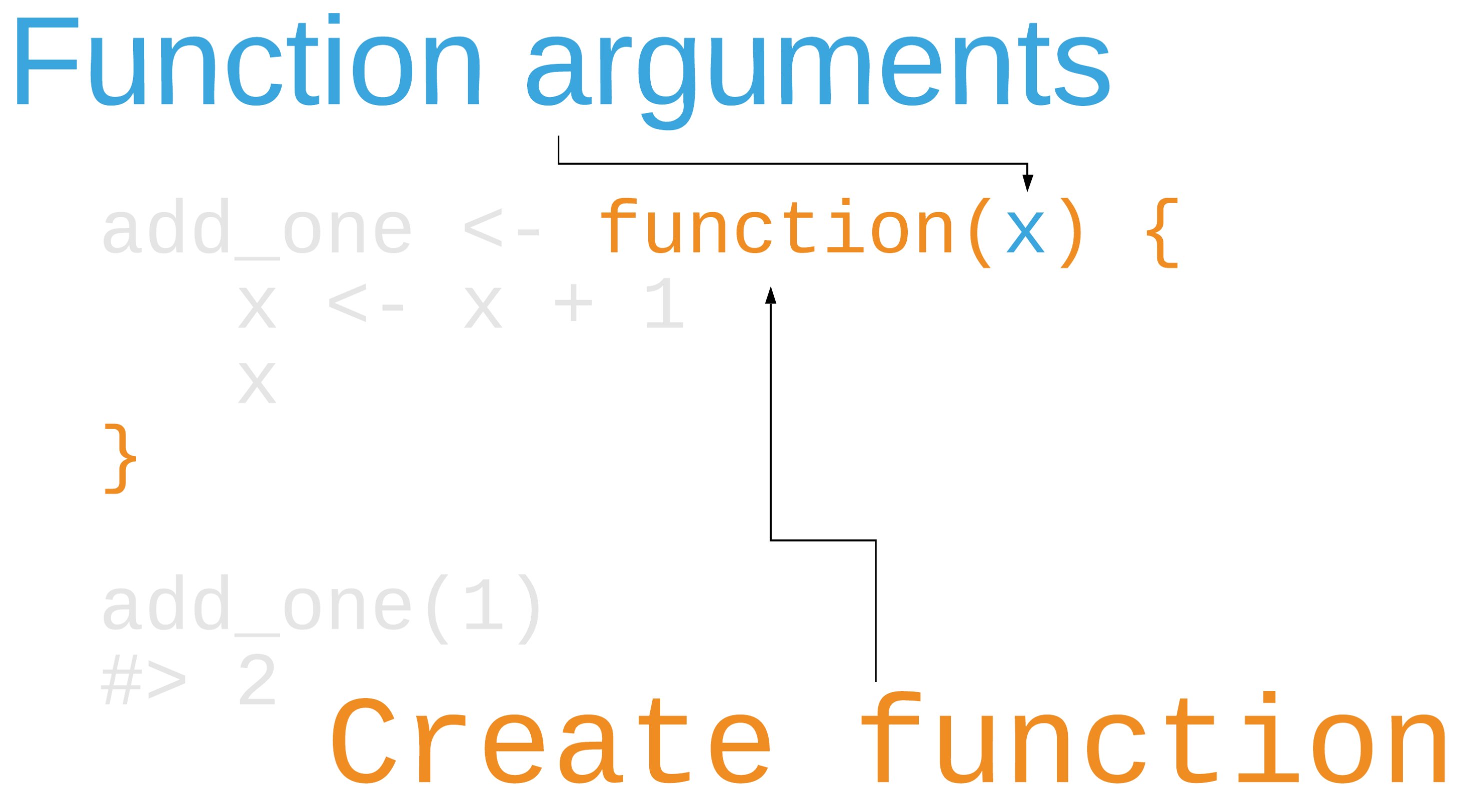
```
## [1] 10
```

Writing Functions

Writing functions

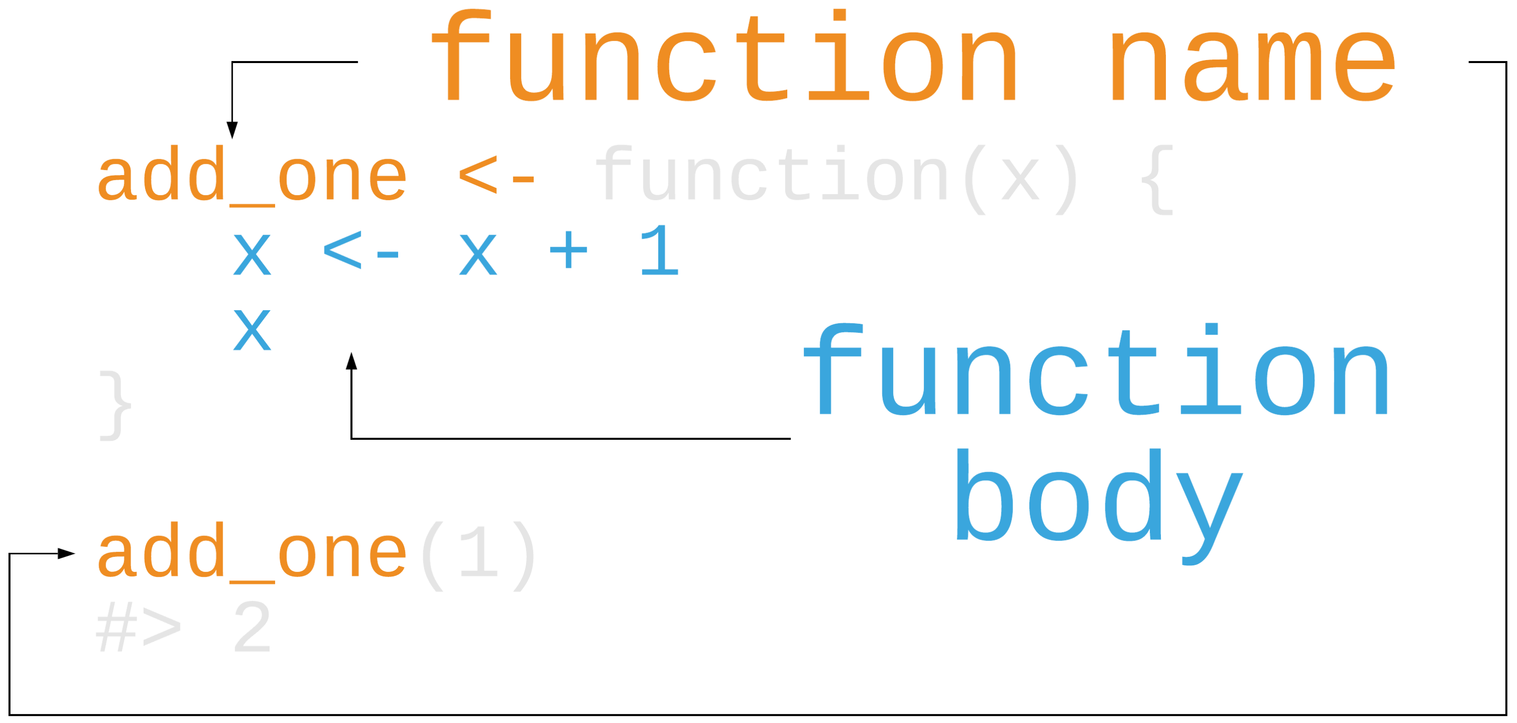
```r
add_one <- function(x) {
   x <- x + 1
   x
}

add_one(1)
#> 2
```

Writing functions



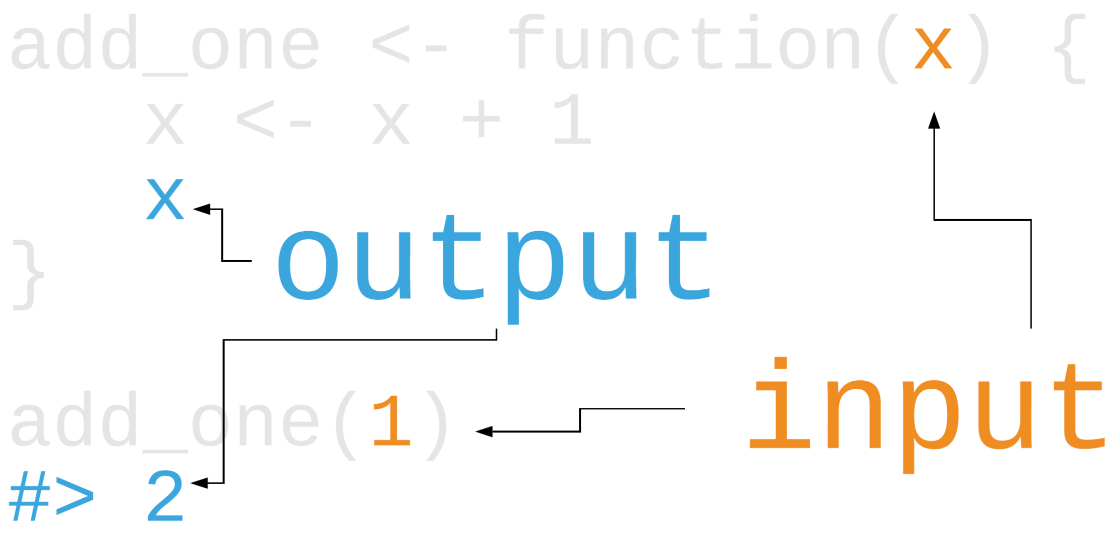
```
add_one <- function(x) {
    x <- x + 1
    x
}

add_one(1)
#> 2
```

Writing functions



```r
add_one <- function(x) {
    x <- x + 1
    x
}

add_one(1)
#> 2
```

Writing functions



```
add_one <- function(x) {
    x <- x + 1
    x
}

add_one(1)
#> 2
```

Your Turn 3

Create a function called `sim_data` **that doesn't take any arguments.**

In the function body, we'll return a `tibble`**.**

For `x`**, have** `rnorm()` **return 50 random numbers.**

For `sex`**, use** `rep()` **to create 50 values of "male" and "female". Hint: You'll have to give** `rep()` **a character vector. for the first argument. The** `times` **argument is how many times** `rep()` **should repeat the first argument, so make sure you account for that.**

For `age()` **use the** `sample()` **function to sample 50 numbers from 25 to 50 with replacement.**

Call `sim_data()`

Your Turn 3

```r
sim_data <- function() {
  tibble(
    x = rnorm(50),
    sex = rep(c("male", "female"), times = 25),
    age = sample(25:50, size = 50, replace = TRUE)
  )
}

sim_data()
```

Your Turn 3

```r
sim_data <- function() {
  tibble(
    x = rnorm(50),
    sex = rep(c("male", "female"), times = 25),
    age = sample(25:50, size = 50, replace = TRUE)
  )
}

sim_data()
```

Your Turn 3

```r
sim_data <- function() {
  tibble(
    x = rnorm(50),
    sex = rep(c("male", "female"), times = 25),
    age = sample(25:50, size = 50, replace = TRUE)
  )
}

sim_data()
```

Your Turn 3

```r
sim_data <- function() {
  tibble(
    x = rnorm(50),
    sex = rep(c("male", "female"), times = 25),
    age = sample(25:50, size = 50, replace = TRUE)
  )
}

sim_data()
```

Your Turn 3

```
## # A tibble: 50 x 3
##          x sex      age
##      <dbl> <chr>  <int>
##  1 -0.434  male      50
##  2 -0.680  female    36
##  3 -0.0821 male      31
##  4  1.19   female    42
##  5  0.839  male      31
##  6 -0.838  female    27
##  7  1.61   male      25
##  8 -0.511  female    25
##  9  1.24   male      37
## 10  0.169  female    40
## # … with 40 more rows
```

E-Values

The strength of unmeasured confounding required to explain away a value

E-Values

The strength of unmeasured confounding required to explain away a value

Rate ratio: 3.9 = E-value: 7.3

Your Turn 4

Write a function to calculate an E-Value given an RR.

Call the function evalue **and give it an argument called** estimate**. In the body of the function, calculate the E-Value using** estimate + sqrt(estimate * (estimate - 1))

Call evalue() **for a risk ratio of 3.9**

Your Turn 4

```r
evalue <- function(estimate) {
  estimate + sqrt(estimate * (estimate - 1))
}
```

```r
evalue(3.9)
```

```
## [1] 7.263034
```

Control Flow

```
if (PREDICATE) {
  true_result
}
```

```
if (PREDICATE) {
  true_result
} else {
  default_result
}
```

```
if (PREDICATE) {
  true_result
} else if (ANOTHER_PREDICATE) {
  true_result
} else  {
  default_result
}
```

Other functions to control flow

```r
ifelse(PREDICATE, true_result, false_result)
dplyr::case_when(
  PREDICATE ~ true_result,
  PREDICATE ~ true_result,
  TRUE ~ default_result
)
switch(
  x,
  value1 = result,
  value2 = result
)
```

Validation and stopping

if (is.numeric(x))

stop(), warn()

```r
function(x) {
  if (is.numeric(x)) stop("x must be a character")
  # do something with a character
}
```

Your Turn 5

Use `if ()` **together with** `is.numeric()` **to make sure** `estimate` **is a number. Remember to use** `!` **for not.**

If the estimate is less than 1, set `estimate` **to be equal to** `1 / estimate`**.**

Call `evalue()` **for a risk ratio of 3.9. Then try 0.80. Then try a character value.**

Your Turn 5

```r
evalue <- function(estimate) {
  if (!is.numeric(estimate)) stop("`estimate` must be numeric")
  if (estimate < 1) estimate <- 1 / estimate
  estimate + sqrt(estimate * (estimate - 1))
}
```

Your Turn 5

```r
evalue(3.9)
```

```
## [1] 7.263034
```

```r
evalue(.80)
```

```
## [1] 1.809017
```

```r
evalue("3.9")
```

```
## Error in evalue("3.9"): `estimate` must be numeric
```

Your Turn 6

Add a new argument called `type`**. Set the default value to "rr"**

Check if `type` **is equal to "or". If it is, set the value of** `estimate` **to be** `sqrt(estimate)`

Call `evalue()` **for a risk ratio of 3.9. Then try it again with** `type = "or"`.

Your Turn 6

```r
evalue <- function(estimate, type = "rr") {
  if (!is.numeric(estimate)) stop("`estimate` must be numeric")
  if (type == "or") estimate <- sqrt(estimate)
  if (estimate < 1) estimate <- 1 / estimate
  estimate + sqrt(estimate * (estimate - 1))
}
```

Your Turn 6

```r
evalue(3.9)
```

```
## [1] 7.263034
```

```r
evalue(3.9, type = "or")
```

```
## [1] 3.362342
```

Your Turn 7: Challenge!

Create a new function called transform_to_rr **with arguments** estimate **and** type**.**

Use the same code above to check if type == "or" **and transform if so. Add another line that checks if** type == "hr"**. If it does, transform the estimate using this formula:** (1 - 0.5^sqrt(estimate)) / (1 - 0.5^sqrt(1 / estimate))**.**

Move the code that checks if estimate < 1 **to** transform_to_rr **(below the OR and HR transformations)**

Return estimate

In evalue()**, change the default argument of** type **to be a character vector containing "rr", "or", and "hr".**

Get and validate the value of type **using** match.arg()**. Follow the pattern** argument_name <- match.arg(argument_name)

Transform estimate **using** transform_to_rr()**. Don't forget to pass it both** estimate **and** type**!**

Your Turn 7: Challenge!

```r
transform_to_rr <- function(estimate, type) {
  if (type == "or") estimate <- sqrt(estimate)
  if (type == "hr") {
    estimate <-
      (1 - 0.5^sqrt(estimate)) / (1 - 0.5^sqrt(1 / estimate))
  }
  if (estimate < 1) estimate <- 1 / estimate

  estimate
}

evalue <- function(estimate, type = c("rr", "or", "hr")) {
  # validate arguments
  if (!is.numeric(estimate)) stop("`estimate` must be numeric")
  type <- match.arg(type)

  # calculate evalue
  estimate <- transform_to_rr(estimate, type)
  estimate + sqrt(estimate * (estimate - 1))
}
```

Your Turn 7: Challenge!

```r
transform_to_rr <- function(estimate, type) {
  if (type == "or") estimate <- sqrt(estimate)
  if (type == "hr") {
    estimate <-
      (1 - 0.5^sqrt(estimate)) / (1 - 0.5^sqrt(1 / estimate))
  }
  if (estimate < 1) estimate <- 1 / estimate

  estimate
}

evalue <- function(estimate, type = c("rr", "or", "hr")) {
  # validate arguments
  if (!is.numeric(estimate)) stop("`estimate` must be numeric")
  type <- match.arg(type)

  # calculate evalue
  estimate <- transform_to_rr(estimate, type)
  estimate + sqrt(estimate * (estimate - 1))
}
```

Your Turn 7: Challenge!

```r
transform_to_rr <- function(estimate, type) {
  if (type == "or") estimate <- sqrt(estimate)
  if (type == "hr") {
    estimate <-
      (1 - 0.5^sqrt(estimate)) / (1 - 0.5^sqrt(1 / estimate))
  }
  if (estimate < 1) estimate <- 1 / estimate

  estimate
}

evalue <- function(estimate, type = c("rr", "or", "hr")) {
  # validate arguments
  if (!is.numeric(estimate)) stop("`estimate` must be numeric")
  type <- match.arg(type)

  # calculate evalue
  estimate <- transform_to_rr(estimate, type)
  estimate + sqrt(estimate * (estimate - 1))
}
```

Your Turn 7: Challenge!

```r
evalue(3.9)
```

```
## [1] 7.263034
```

```r
evalue(3.9, type = "or")
```

```
## [1] 3.362342
```

```r
evalue(3.9, type = "hr")
```

```
## [1] 4.474815
```

```r
evalue(3.9, type = "rd")
```

```
## Error in match.arg(type): 'arg' should be one of "rr", "or", "hr"
```

Pass the dots: ...

```
select_gapminder <- function(...) {
  gapminder %>%
    select(...)
}

select_gapminder(pop, year)
```

Pass the dots: ...

```r
select_gapminder <- function(...) {
  gapminder %>%
    select(...)
}

select_gapminder(pop, year)
```

Pass the dots: ...

```
## # A tibble: 1,704 x 2
##          pop  year
##        <int> <int>
##  1   8425333  1952
##  2   9240934  1957
##  3  10267083  1962
##  4  11537966  1967
##  5  13079460  1972
##  6  14880372  1977
##  7  12881816  1982
##  8  13867957  1987
##  9  16317921  1992
## 10  22227415  1997
## # … with 1,694 more rows
```

Your Turn 8

Use ... **to pass the arguments of your function,** filter_summarize()**, to** filter()**.**

In summarize, get the n and mean life expectancy for the data set

Check filter_summarize() **with** year == 1952**.**

Try filter_summarize() **again for 2002, but also filter countries that have "and" in the country name. Use** str_detect() **from the stringr package.**

Your Turn 8

```
filter_summarize <- function(...) {
  gapminder %>%
    filter(...) %>%
    summarize(n = n(), mean_lifeExp = mean(lifeExp))
}
```

```r
filter_summarize(year == 1952)
```

```
## # A tibble: 1 x 2
##       n mean_lifeExp
##   <int>        <dbl>
## 1   142         49.1
```

```r
filter_summarize(year == 2002, str_detect(country, " and "))
```

```
## # A tibble: 1 x 2
##       n mean_lifeExp
##   <int>        <dbl>
## 1     4         69.9
```

Programming with dplyr, ggplot2, and friends

```r
plot_hist <- function(x) {
  ggplot(gapminder, aes(x = x)) + geom_histogram()
}
```

Programming with dplyr, ggplot2, and friends

```r
plot_hist <- function(x) {
  ggplot(gapminder, aes(x = x)) + geom_histogram()
}
```

```r
plot_hist(lifeExp)
```

```
## Error in FUN(X[[i]], ...): object 'lifeExp' not found
```

Programming with dplyr, ggplot2, and friends

```r
plot_hist <- function(x) {
  ggplot(gapminder, aes(x = x)) + geom_histogram()
}
```

```r
plot_hist("lifeExp")
```

```
## Error: StatBin requires a continuous x variable: the x variable is di
```

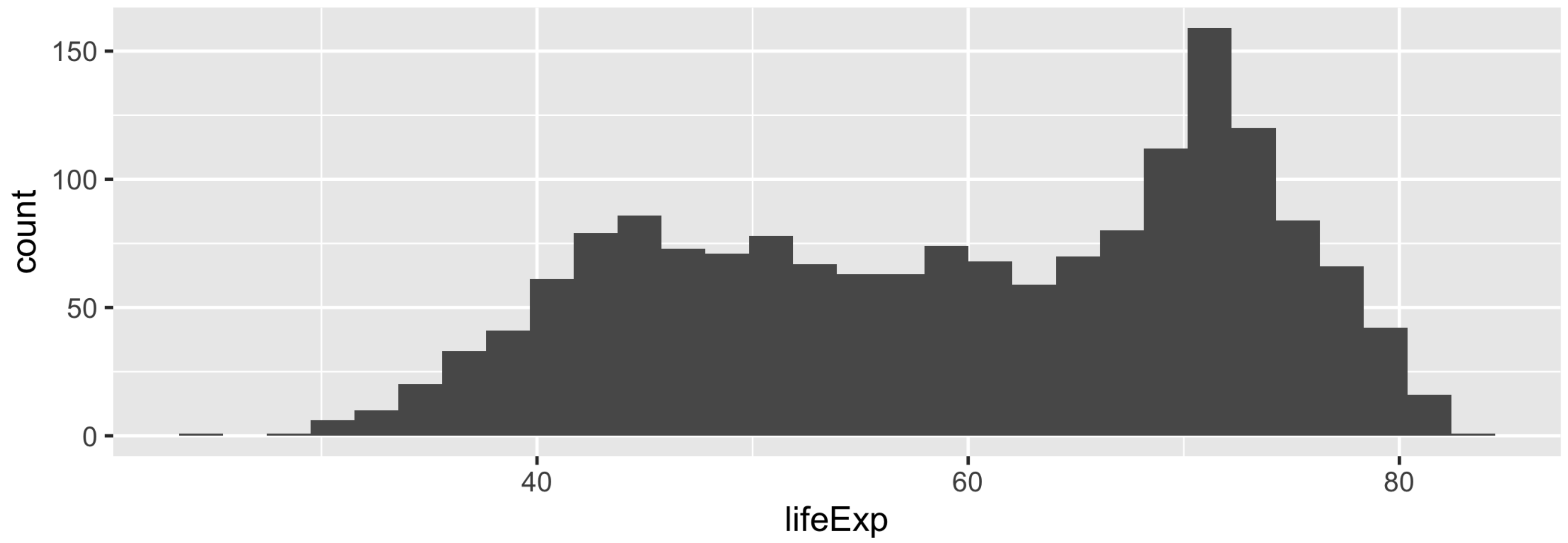
Curly-curly

```
plot_hist <- function(x) {
  ggplot(gapminder, aes(x = {{x}})) + geom_histogram()
}
```

Curly-curly

```r
plot_hist <- function(x) {
  ggplot(gapminder, aes(x = {{x}})) + geom_histogram()
}
```

```r
plot_hist(lifeExp)
```



Your turn 9

Filter gapminder by year **using the value of** .year **(notice the period before hand!). You do NOT need curly-curly for this. (Why is that?)**

Arrange it by the variable. This time, do wrap it in curly-curly!

Make a scatter plot. Use variable **for x. For y, we'll use** country**, but to keep it in the order we arranged it by, we'll turn it into a factor. Wrap the the** factor() **call with** fct_inorder()**. Check the help page if you want to know more about what this is doing.**

Your turn 9

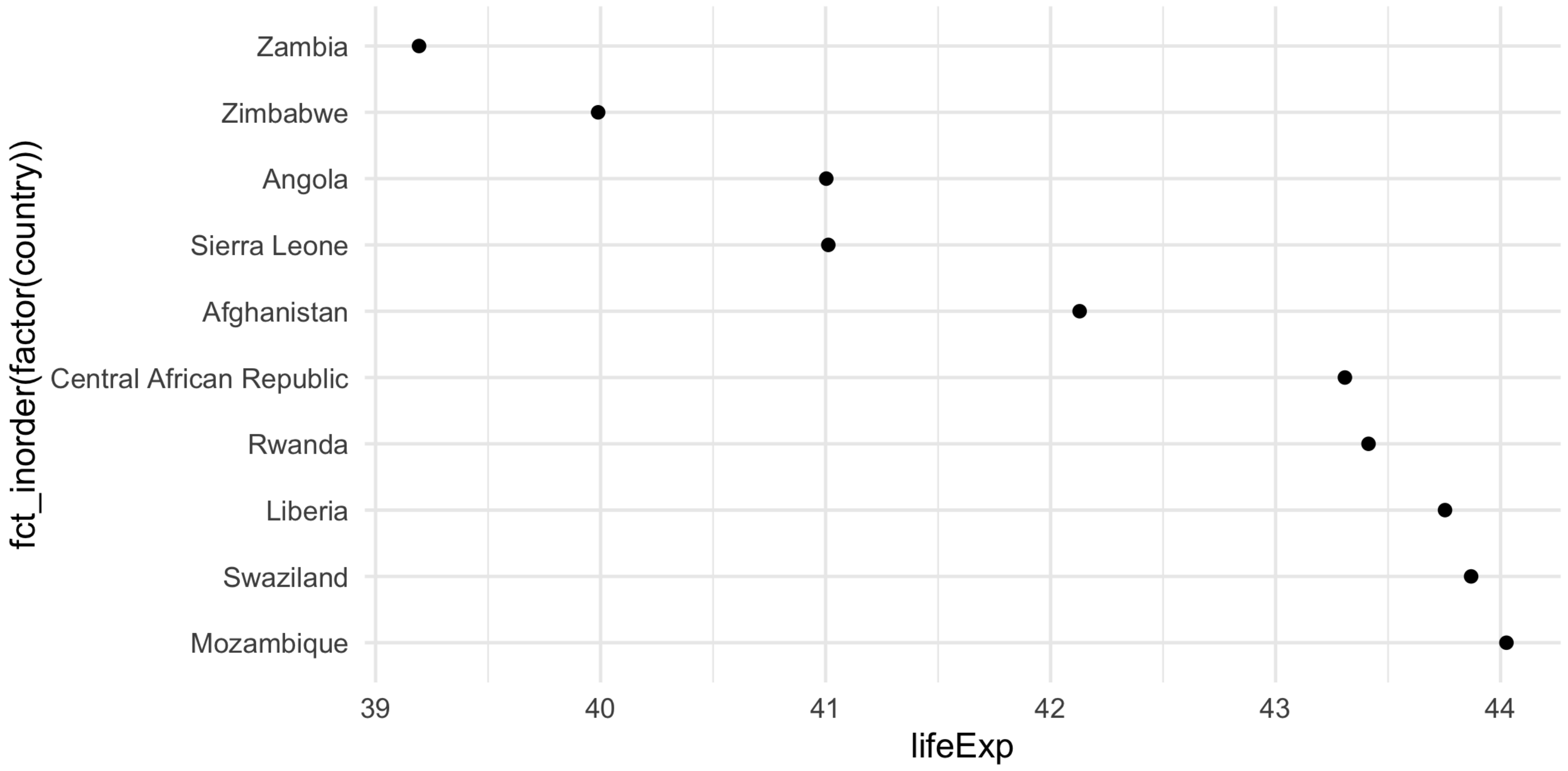
```r
top_scatter_plot <- function(variable, .year) {
  gapminder %>%
    filter(year == .year) %>%
    arrange(desc({{variable}})) %>%
    #  take the 10 lowest values
    tail(10) %>%
    ggplot(aes(x = {{variable}}, y = fct_inorder(factor(country))))
    geom_point() +
    theme_minimal()
}
```

Your turn 9

```r
top_scatter_plot <- function(variable, .year) {
  gapminder %>%
    filter(year == .year) %>%
    arrange(desc({{variable}})) %>%
    #  take the 10 lowest values
    tail(10) %>%
    ggplot(aes(x = {{variable}}, y = fct_inorder(factor(country))))
    geom_point() +
    theme_minimal()
}
```

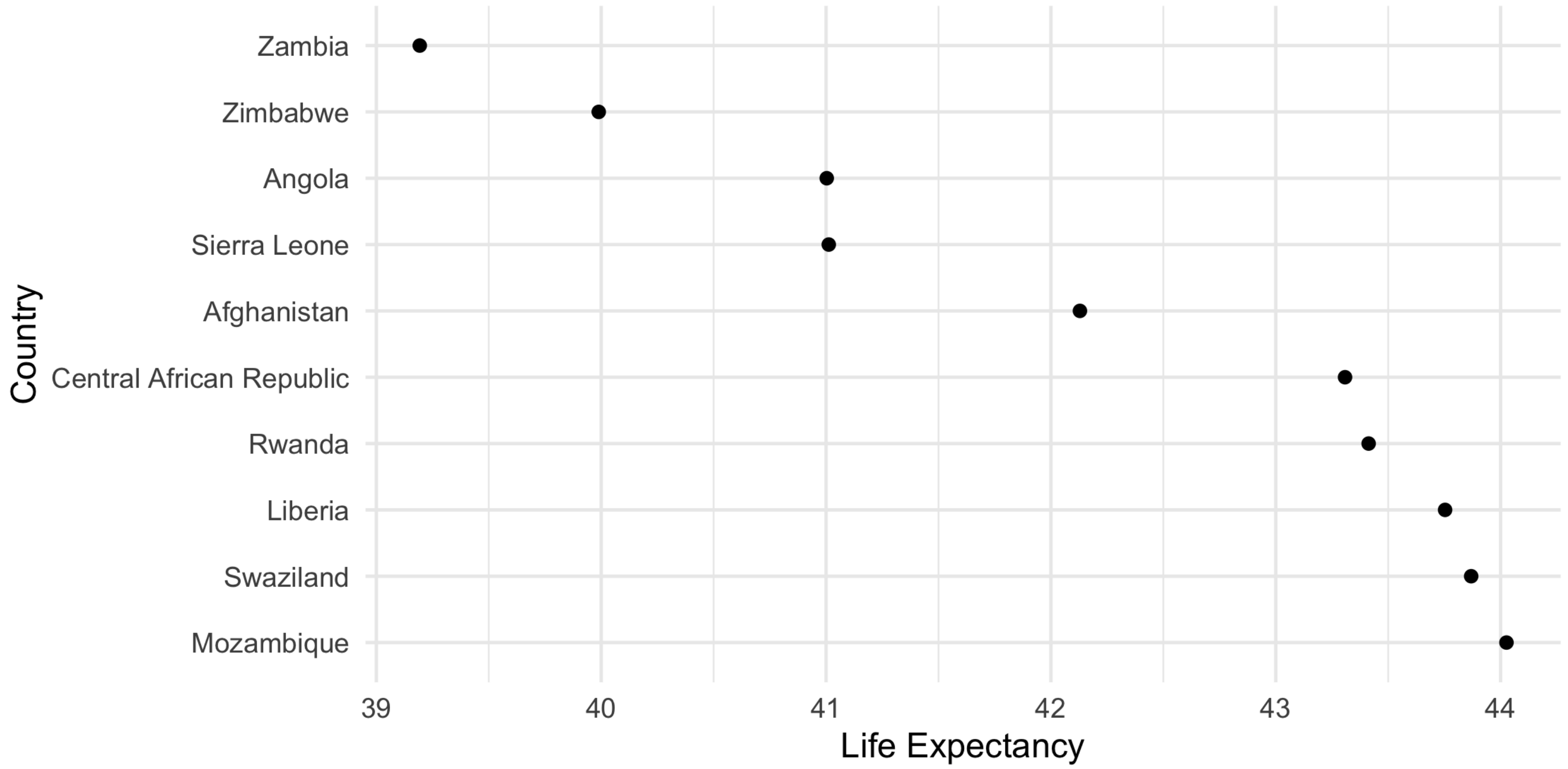
Your turn 9

```r
top_scatter_plot(lifeExp, 2002)
```



Your turn 9

```r
top_scatter_plot(lifeExp, 2002) +
  labs(x = "Life Expectancy", y = "Country")
```



Resources

R for Data Science: A comprehensive but friendly introduction to the tidyverse. Free online.

Advanced R, 2nd ed.: Detailed guide to how R works and how to make your code better. Free online.

RStudio Primers: Free interactive courses in the Tidyverse