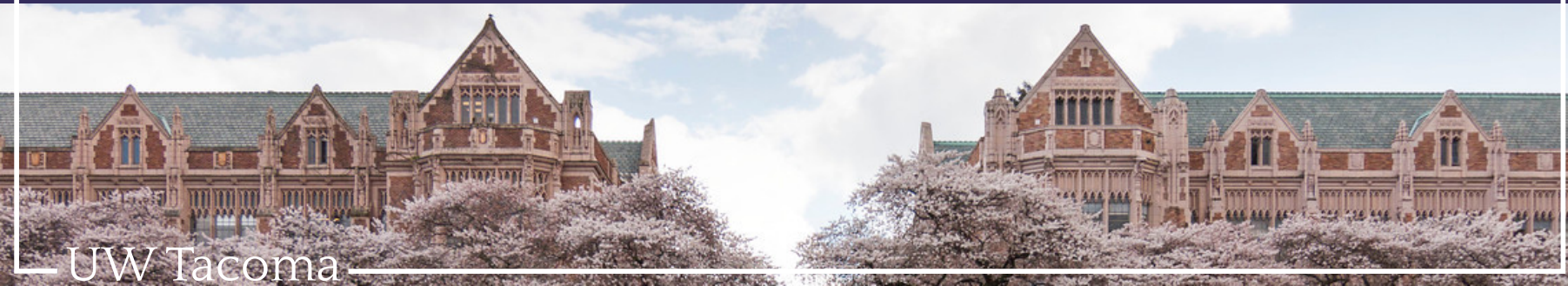# R Exposure 3

Unit 2: Vectorization and Functions

Chuck Lanfear

Apr 19, 2020
Updated: Apr 18, 2020

UW Tacoma

A Quick Aside

Visualize the Goal First

Before you can write effective code, you need to know *exactly* what you want that code to produce.

- Do I want a single value? A vector? List?
- Do I want one observation per person? Person-year? Year?

Most programming problems can be reduced to having an unclear idea of your end **goal** (or your beginning state).

If you know what you *have* (the data structure) and what you *want*, the intermediate steps are usually obvious.

When in doubt, *sketch* the beginning state and the intended end state. Then consider what translates the former into the latter in the least complicated way.

If that seems complex, break it into more steps.

Vectorization

Example from Last Part

Remember when we tried find the mean for each variable in the `swiss` data?

The best solution is to just use `colMeans()` without even thinking about pre-allocation or `for()` loops:

```
colMeans(swiss)
```

```
##         Fertility      Agriculture       Examination         Education
##              70.1             50.7              16.5              11.0
##          Catholic  Infant.Mortality
##              41.1             19.9
```

Vectorization Avoids Loops

Loops are very powerful and applicable in almost any situation.

They are also slow and require writing more code than vectorized commands.

Whenever possible, use existing vectorized commands like `colMeans()` or `dplyr` functions.

Sometimes no functions exist to do what you need, so you'll be tempted to write a loop.

This makes sense on a *fast, one-time operation, on small data.*

If your data are large or you're going to do it repeatedly, however, consider *writing your own functions*!

Writing Functions

Examples of Existing Functions

- `mean()`:
  - Input: a vector
  - Output: a single number

- `dplyr::filter()`:
  - Input: a data frame, logical conditions
  - Output: a data frame with rows removed using those conditions

- `readr::read_csv()`:
  - Input: a file path, optionally variable names or types
  - Output: a data frame containing info read in from file

Why Write Your Own Functions?

Functions can encapsulate repeated actions such as:

- Given a vector, compute some special summary stats
- Given a vector and definition of "invalid" values, replace with `NA`
- Templates for favorite `ggplot`s used in reports

Advanced function applications (not covered in this class):

- Parallel processing
- Generating *other* functions
- Making custom packages containing your functions

Simple Function

Let's look at a function that takes a vector as input and outputs a named vector of the first and last elements:

```r
first_and_last <- function(x) {
    first <- x[1]
    last  <- x[length(x)]
    return(c("first" = first, "last" = last))
}
```

Test it out:

```r
first_and_last(c(4, 3, 1, 8))
```

```
## first  last
##    4     8
```

Testing `first_and_last`

What if I give `first_and_last()` a vector of length 1?

```
first_and_last(7)
```

```
## first  last
##     7     7
```

Of length 0?

```
first_and_last(numeric(0))
```

```
## first
##    NA
```

Maybe we want it to be a little smarter.

Checking Inputs

Let's make sure we get an error message when the vector is too small:

```r
smarter_first_and_last <- function(x) {
    if(length(x) == 0L) { # specify integers with L
        stop("The input has no length!")
    } else {
        first <- x[1]
        last  <- x[length(x)]
        return(c("first" = first, "last" = last))
    }
}
```

`stop()` ceases running the function and prints the text inside as an error message.

Testing Smarter Function

```
smarter_first_and_last(numeric(0))
```

```
## Error in smarter_first_and_last(numeric(0)): The input has no length!
```

```
smarter_first_and_last(c(4, 3, 1, 8))
```

```
## first  last
##     4     8
```

Cracking Open Functions

If you type a function name without any parentheses or arguments, you can see its contents:

```
smarter_first_and_last
```

```
## function(x) {
##     if(length(x) == 0L) { # specify integers with L
##         stop("The input has no length!") #<<
##     } else {
##         first <- x[1]
##         last  <- x[length(x)]
##         return(c("first" = first, "last" = last))
##     }
## }
## <environment: 0x0000021dcfd64910>
```

You can also put your cursor over a function in your syntax and hit `F2` .

Anatomy of a Function

```
NAME <- function(ARGUMENT1, ARGUMENT2=DEFAULT){
  BODY
  return(OUTPUT)
}
```

- **Name**: What you assign the function to so you can use it later
  - You can have "anonymous" (no-name) functions
- **Arguments** (aka inputs, parameters): things the user passes to the function that affect how it works
  - e.g. `x` or `na.rm` in `function(x, na.rm = FALSE){...}`
  - `na.rm = FALSE` is example of setting a default value: if user doesn't say what `na.rm` is, it'll be `FALSE`
  - `x`, `na.rm` values won't exist in R outside of the function
- **Body**: The actual operations inside the function.
- **Return Value**: The output inside `return()`. Could be a vector, list, data frame, another function, or even nothing
  - If unspecified, will be the last thing calculated (maybe not what you want?)

Example: Reporting Quantiles

Maybe you want to know more detailed quantile information than `summary()` gives you and with interpretable names.

Here's a starting point:

```r
quantile_report <- function(x, na.rm = FALSE) {
    quants <- quantile(x, na.rm = na.rm,
        probs = c(0.01, 0.05, 0.10, 0.25, 0.5, 0.75, 0.90, 0.95, 0.99))
    names(quants) <- c("Bottom 1%", "Bottom 5%", "Bottom 10%", "Bottom 25%",
                        "Median", "Top 25%", "Top 10%", "Top 5%", "Top 1%")
    return(quants)
}
quantile_report(rnorm(10000))
```

```
##  Bottom 1%  Bottom 5% Bottom 10% Bottom 25%     Median    Top 25%
##   -2.33024   -1.65438   -1.28137   -0.67938   -0.00764    0.66105
##     Top 10%     Top 5%     Top 1%
##     1.26276    1.63364    2.37578
```

An Aside on Apply functions

`lapply()`: List + Functions

`lapply()` is used to **apply** a function over a **l**ist of any kind (e.g. a data frame) and return a list. This is a lot easier than preparing a `for()` loop!

```
lapply(swiss, FUN = quantile_report)
```

```
## $Fertility
##  Bottom 1%  Bottom 5% Bottom 10% Bottom 25%     Median    Top 25%
##       38.6       47.6       56.2       64.7       70.4       78.4
##     Top 10%      Top 5%      Top 1%
##       84.6       90.7       92.5
##
## $Agriculture
##  Bottom 1%  Bottom 5% Bottom 10% Bottom 25%     Median    Top 25%
##       4.19      15.65      17.36      35.90      54.10      67.65
##     Top 10%      Top 5%      Top 1%
##      76.82      84.81      87.95
##
## $Examination
##  Bottom 1%  Bottom 5% Bottom 10% Bottom 25%     Median    Top 25%
##        3.0        5.0        6.0       12.0       16.0       22.0
##     Top 10%      Top 5%      Top 1%
##       26.0       30.4       36.1
```

`sapply()`: Simplified `lapply()`

A downside to `lapply()` is that lists are complicated. `sapply()` **s**implifies output by into a matrix or vector:

```
sapply(swiss, FUN = quantile_report)
```

```
##              Fertility Agriculture Examination Education Catholic
## Bottom 1%       38.6        4.19         3.0      1.46     2.21
## Bottom 5%       47.6       15.65         5.0      2.00     2.45
## Bottom 10%      56.2       17.36         6.0      3.00     2.83
## Bottom 25%      64.7       35.90        12.0      6.00     5.20
## Median          70.4       54.10        16.0      8.00    15.14
## Top 25%         78.4       67.65        22.0     12.00    93.12
## Top 10%         84.6       76.82        26.0     23.20    99.00
## Top 5%          90.7       84.81        30.4     29.00    99.61
## Top 1%          92.5       87.95        36.1     43.34    99.87
##              Infant.Mortality
## Bottom 1%            12.8
## Bottom 5%            15.6
## Bottom 10%           16.4
## Bottom 25%           18.1
## Median               20.0
## Top 25%              21.7
## Top 10%              23.7
## Top 5%               24.5
## Top 1%               25.8
```

`apply()`

There is also `apply()` which works over matrices or data frames. You can apply the function to each row (`MARGIN = 1`) or column (`MARGIN = 2`).

```
apply(swiss, MARGIN = 2, FUN = quantile_report)
```

```
##            Fertility Agriculture Examination Education Catholic
## Bottom 1%       38.6        4.19         3.0      1.46     2.21
## Bottom 5%       47.6       15.65         5.0      2.00     2.45
## Bottom 10%      56.2       17.36         6.0      3.00     2.83
## Bottom 25%      64.7       35.90        12.0      6.00     5.20
## Median          70.4       54.10        16.0      8.00    15.14
## Top 25%         78.4       67.65        22.0     12.00    93.12
## Top 10%         84.6       76.82        26.0     23.20    99.00
## Top 5%          90.7       84.81        30.4     29.00    99.61
## Top 1%          92.5       87.95        36.1     43.34    99.87
##            Infant.Mortality
## Bottom 1%              12.8
## Bottom 5%              15.6
## Bottom 10%             16.4
## Bottom 25%             18.1
## Median                 20.0
## Top 25%                21.7
## Top 10%                23.7
## Top 5%                 24.5
## Top 1%                 25.8
```

Vectorized Data Loading

Remember the loop for loading data files from last unit?

```r
library(dplyr); library(readr)
file_list  <- list.files("./example_data/")
file_paths <- paste0("./example_data/", file_list)
data_names <- stringr::str_remove(file_list, ".csv")
data_list  <- vector("list", length(file_list))
names(data_list) <- data_names
for (i in seq_along(file_list)){
  data_list[[ data_names[i] ]] <- read_csv(file_paths[i])
}
complete_data <- bind_rows(data_list)
head(complete_data, 3)
```

```
## # A tibble: 3 x 3
##      id      x      z
##   <dbl>  <dbl> <dbl>
## 1    44  0.516 0.381
## 2    49  2.17  0.346
## 3    50 -0.122 0.711
```

Vectorized Data Loading

Another way to load these files would be to... `lapply()` over the file names then bind the rows together. Faster and easier![1]

```
complete_data <- lapply(file_paths, read_csv) %>%
  bind_rows()
head(complete_data, 3)
```

```
## # A tibble: 3 x 3
##      id      x      z
##   <dbl>  <dbl>  <dbl>
## 1    44  0.516  0.381
## 2    49  2.17   0.346
## 3    50 -0.122  0.711
```

Most loops can be replaced by a simpler `*apply()` call.

- No pre-allocation
- *Usually* much faster

[1] *Even faster and easier* would be to use `vroom::vroom()` on the directory.

Back to Making functions!

Example: Discretizing Continuous Data

Maybe you often want to bucket variables in your data into groups based on quantiles:

Person	Income	Income Bucket
1	8000	1
2	103000	3
3	12000	1
4	52000	2
5	150000	3
6	45000	2

UW Tacoma

Bucketing Function

There's already a function in R called `cut()` that does this, but you need to tell it breaks or the number of buckets.

Let's make a function that calls `cut()` using quantiles (`quants`) for splitting and returns integers:

```r
bucket <- function(x, quants = c(0.333, 0.667)) {
    # set low extreme, quantile points, high extreme
    new_breaks <- c(min(x)-1, quantile(x, probs = quants), max(x)+1)
    # labels = FALSE will return integer codes instead of ranges
    return(cut(x, breaks = new_breaks, labels = FALSE))
}
```

By default this will produce *three buckets*:

1. Anything below 33.3rd percentile
2. Anthing from 33.3rd to 66.7th
3. Anything above 66.7th

To capture all high/low values, we start with `min(x)-1` and end with `max(x)+1`.

Trying Out `bucket()`

```r
dat <- rnorm(100)
dat_quants <- c(0.05, 0.25, 0.5, 0.75, 0.95)
bucketed_dat <- bucket(dat, quants = dat_quants)
plot(x = bucketed_dat, y = dat, main = "Buckets and values", pch = 16)
abline(h = quantile(dat, dat_quants), lty = "dashed", col = "red")
```

Buckets and values
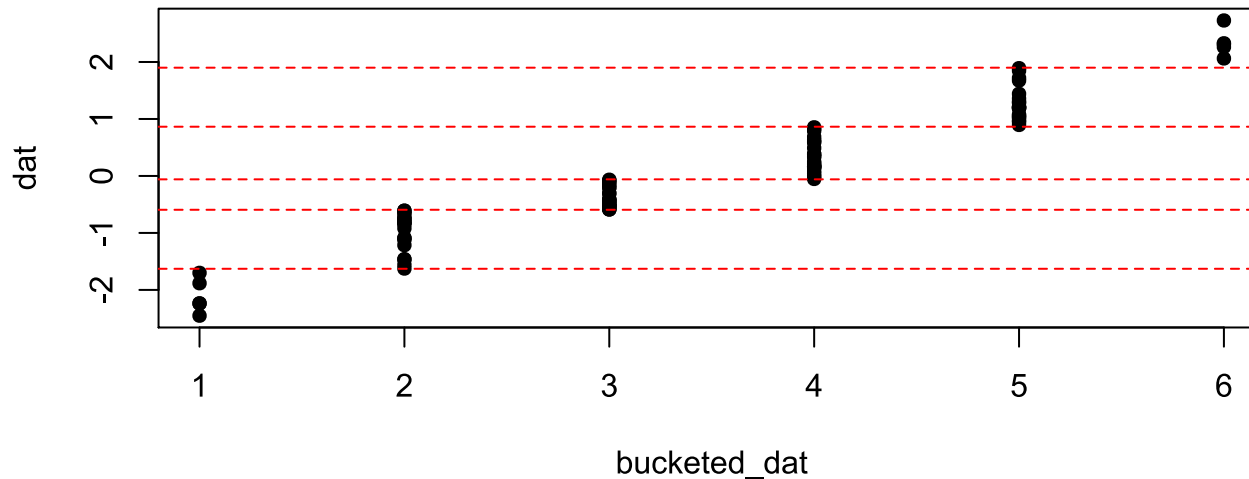
Example: Removing Bad Data

Let's say we have data where impossible values occur:[1]

```r
(school_data <-
  data.frame(school = letters[1:10],
  pr_passing_exam=c(0.78, 0.55, 0.91,   -1, 0.88, 0.81, 0.90, 0.76,   99, 99),
  pr_free_lunch = c(0.33,   99, 0.25, 0.05, 0.12, 0.09, 0.22,  -13, 0.21, 99)))
```

```
##     school pr_passing_exam pr_free_lunch
## 1        a            0.78          0.33
## 2        b            0.55         99.00
## 3        c            0.91          0.25
## 4        d           -1.00          0.05
## 5        e            0.88          0.12
## 6        f            0.81          0.09
## 7        g            0.90          0.22
## 8        h            0.76        -13.00
## 9        i           99.00          0.21
## 10       j           99.00         99.00
```

[1] Different types of missing data are often coded this way in survey and administrative data sets.

Function to Remove Extreme Values

Goal:

- Input: a vector `x`, cutoff for `low`, cutoff for `high`
- Output: a vector with `NA` in the extreme places

```
remove_extremes <- function(x, low, high) {
    x_no_low <- ifelse(x < low, NA, x)
    x_no_low_no_high <- ifelse(x_no_low > high, NA, x)
    return(x_no_low_no_high)
}
remove_extremes(school_data$pr_passing_exam, low = 0, high = 1)
```

```
##  [1] 0.78 0.55 0.91   NA 0.88 0.81 0.90 0.76   NA   NA
```

`dplyr::mutate_at()`

`dplyr` function `mutate_at()` applies functions to multiple variables at once.
We can use this to update every variable (besides `school`) in `school_data`:

```
library(dplyr)
school_data %>%
   mutate_at(vars(-school), ~ remove_extremes(x = ., low = 0, high = 1))
```

```
##    school pr_passing_exam pr_free_lunch
## 1       a            0.78          0.33
## 2       b            0.55            NA
## 3       c            0.91          0.25
## 4       d              NA          0.05
## 5       e            0.88          0.12
## 6       f            0.81          0.09
## 7       g            0.90          0.22
## 8       h            0.76            NA
## 9       i              NA          0.21
## 10      j              NA            NA
```

(Non-)Standard Evaluation

`dplyr` uses what is called **non-standard evaluation** that lets you refer to "naked" variables (no quotes around them) like `school`.

`dplyr` verbs (like `mutate()`) recently started supporting **standard evaluation** allowing you to use quoted object names as well. This makes writing functions and loops with `dplyr` easier.

```
swiss %>%
  select("Fertility", "Catholic") %>%
  head(2)
```

```
##              Fertility Catholic
## Courtelary       80.2     9.96
## Delemont         83.1    84.84
```

Anonymous `dplyr` Functions

You can skip naming your function in `dplyr` if you won't use it again.

For example, this returns the mean divided by the standard deviation for each variable in `swiss`:

```
swiss %>%
    summarize_all( ~ mean(., na.rm = TRUE) / sd(., na.rm = TRUE))
```

```
##   Fertility Agriculture Examination Education Catholic
## 1      5.62        2.23        2.07      1.14    0.987
##   Infant.Mortality
## 1             6.85
```

Anonymous `lapply()`

Like with `dplyr`, you can use anonymous functions in `lapply()`[1], but a difference is you'll need to have the `function()` part at the beginning:

```
lapply(swiss, function(x) mean(x, na.rm = TRUE) / sd(x, na.rm = TRUE))
```

```
## $Fertility
## [1] 5.62
##
## $Agriculture
## [1] 2.23
##
## $Examination
## [1] 2.07
##
## $Education
## [1] 1.14
##
## $Catholic
## [1] 0.987
```

[1] Note that `lapply()` produces a list as output. You could instead use `sapply()` to get a vector.

Example: `ggplot2` Templates

Let's say you have a particular way you like your charts:

```r
library(gapminder); library(ggplot2)
ggplot(gapminder %>% filter(country == "Afghanistan"),
       aes(x = year, y = pop / 1000000)) +
       geom_line(color = "firebrick") +
       xlab(NULL) + ylab("Population (millions)") +
       ggtitle("Population of Afghanistan since 1952") +
       theme_minimal() +
       theme(plot.title = element_text(hjust = 0, size = 20))
```

- How could we make this flexible for any country?
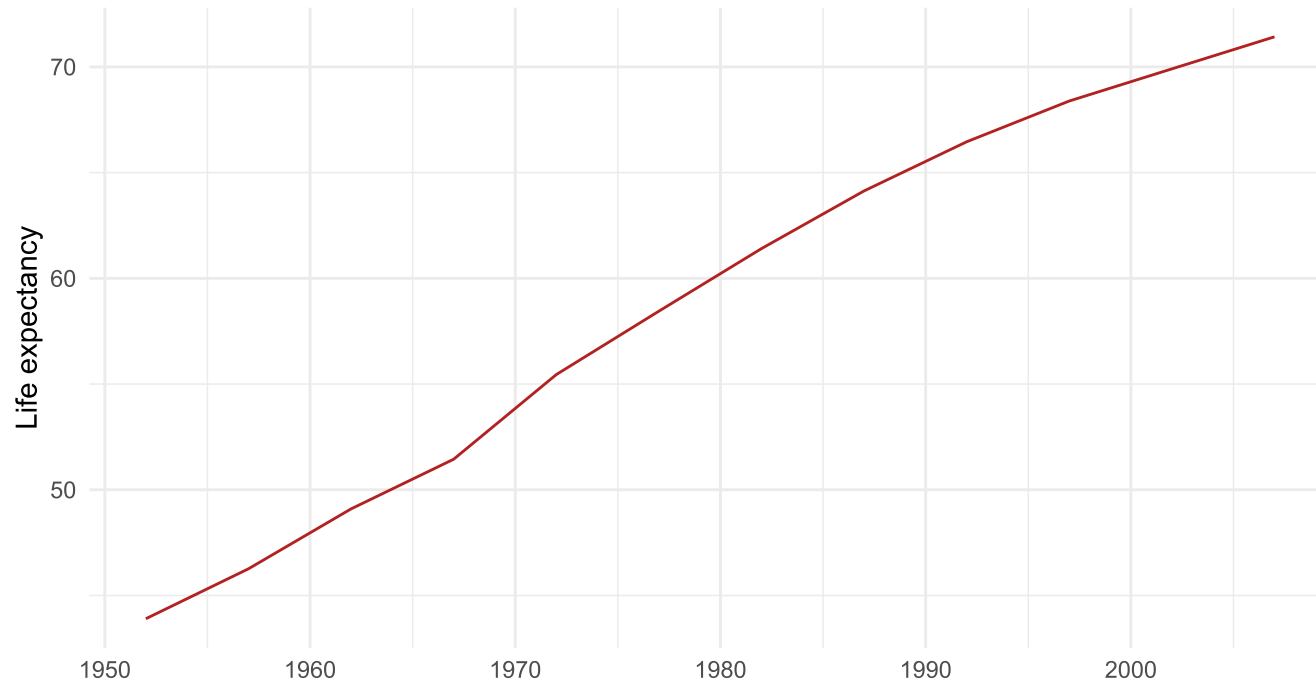- How could we make this flexible for any `gapminder` variable?

Example of Desired Chart

Population of Afghanistan since 1952
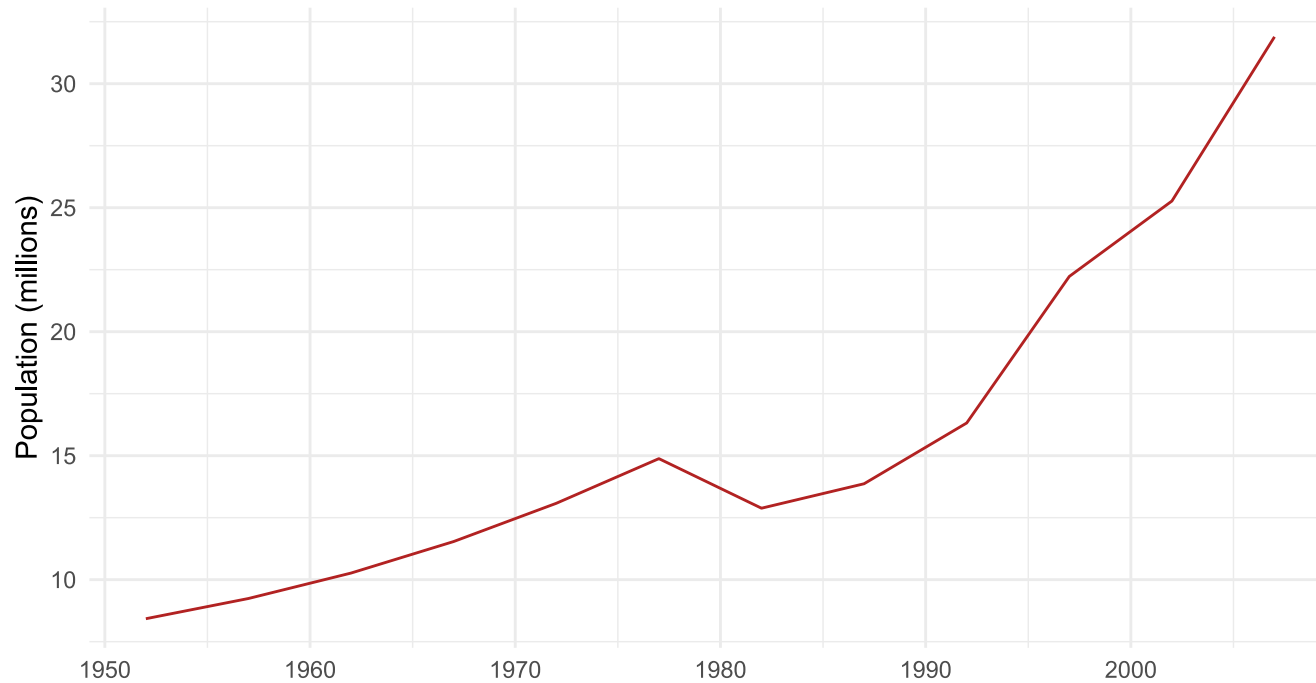
Another Example

Life expectancy in Peru since 1952

UW Tacoma

Making Country Flexible

We can have the user input a character string for `cntry` as an argument to the function to get subsetting and the title right:

```r
gapminder_lifeplot <- function(cntry) {
    ggplot(gapminder %>% filter(country == cntry),
        aes(x = year, y = lifeExp)) +
    geom_line(color = "firebrick") +
    xlab(NULL) + ylab("Life expectancy") + theme_minimal() +
    ggtitle(paste0("Life expectancy in ", cntry, " since 1952")) +
    theme(plot.title = element_text(hjust = 0, size = 20))
}
```
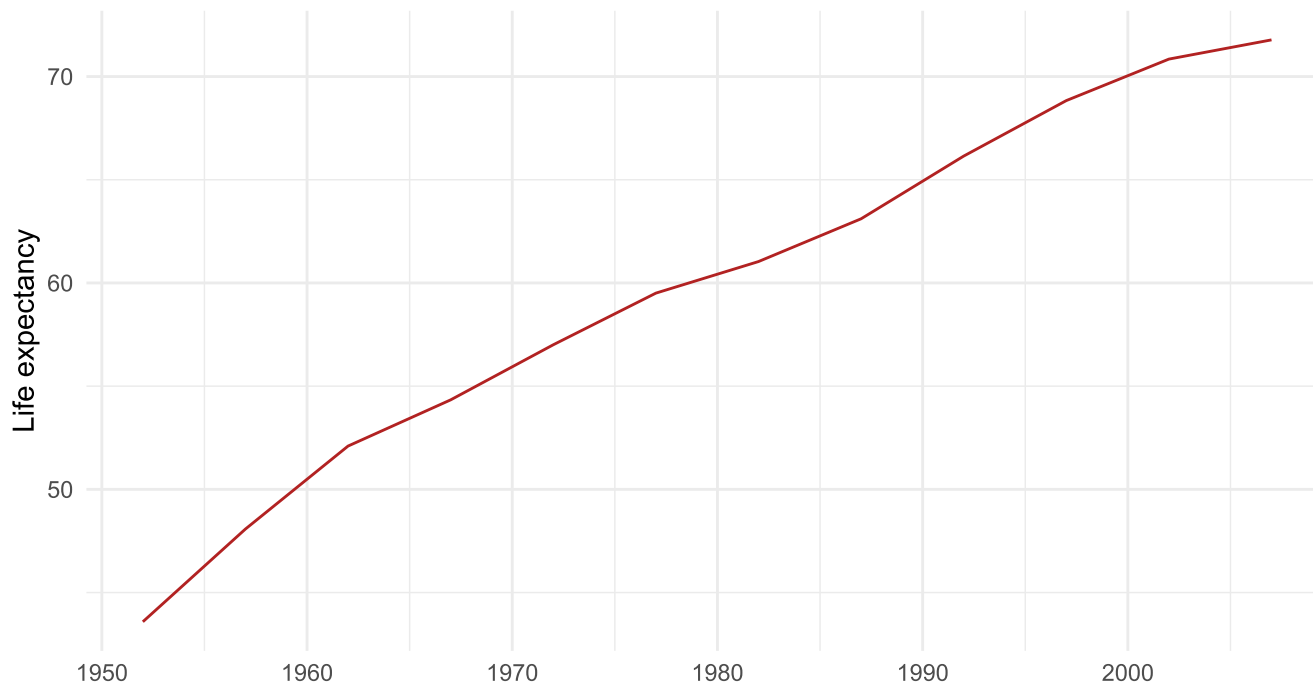
What `cntry` does:

- `filter()` to the specific value of `cntry`

- Add text value of `cntry` in `ggtitle()`

Testing Plot Function

```
gapminder_lifeplot(cntry = "Turkey")
```


Life expectancy in Turkey since 1952

Making y Value Flexible

Now let's allow the user to say which variable they want on the y-axis. How we can get the right labels for the axis and title? We can use a named character vector to serve as a "lookup table" inside the function:

```r
y_axis_label <- c("lifeExp" = "Life expectancy",
                  "pop" = "Population (millions)",
                  "gdpPercap" = "GDP per capita, USD")
title_text <- c("lifeExp" = "Life expectancy in ",
                "pop" = "Population of ",
                "gdpPercap" = "GDP per capita in ")
# example use:
y_axis_label["pop"]
```

```
##                     pop
## "Population (millions)"
```

```r
title_text["pop"]
```

```
##             pop
## "Population of "
```

aes_string()

ggplot() is usually looking for "naked" variables, but we can tell it to take them as quoted strings (standard evaluation) using aes_string() instead of aes(), which is handy when making functions:

```r
gapminder_plot <- function(cntry, yvar) {
    y_axis_label <- c("lifeExp" = "Life expectancy",
                      "pop" = "Population (millions)",
                      "gdpPercap" = "GDP per capita, USD")[yvar]
    title_text   <- c("lifeExp" = "Life expectancy in ",
                      "pop" = "Population of ",
                      "gdpPercap" = "GDP per capita in ")[yvar]
    ggplot(gapminder %>% filter(country == cntry) %>%
             mutate(pop = pop / 1000000),
           aes_string(x = "year", y = yvar)) +
      geom_line(color = "firebrick") +
      ggtitle(paste0(title_text, cntry, " since 1952")) +
      xlab(NULL) + ylab(y_axis_label) + theme_minimal() +
      theme(plot.title = element_text(hjust = 0, size = 20))
}
```

Testing `gapminder_plot()`

```
gapminder_plot(cntry = "Turkey", yvar = "pop")
```

Population of Turkey since 1952
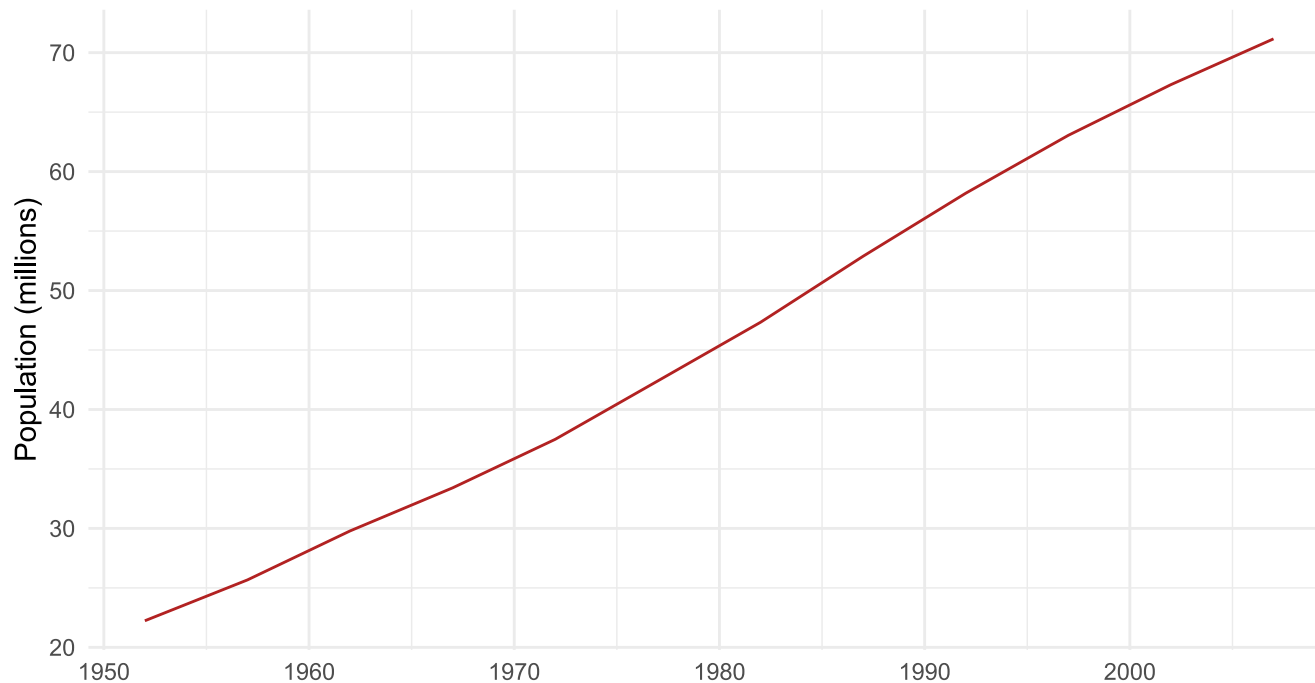
Debugging

Something not working as hoped? Try using `debug()` on a function, which will show you the world as perceived from inside the function:

```
debug(gapminder_plot)
```

Then when you've fixed your problem, use `undebug()` so that you won't go into debug mode every time you run it:

```
undebug(gapminder_plot)
```

Overview: The Process

Data processing can be very complicated, with many valid ways of accomplishing it.

I believe the best general approach is the following:

1. Look carefully at the **starting data** to figure out what you can get from them.
2. Determine *precisely* what you want the **end product** to look like.
3. Identify individual steps needed to go from Step 1 to Step 2.
4. Make each discrete step its own set of functions or function calls.
   - If any step is confusing or complicated, **break it into more steps**.
5. Complete each step *separately and in order*.
   - Do not continue until a step is producing what you need for the next step.
   - **Do not worry about combining steps for efficiency until everything works**.

Once finished, if you need to do this again, *convert the prior steps into functions*!

End of R3, Unit 2