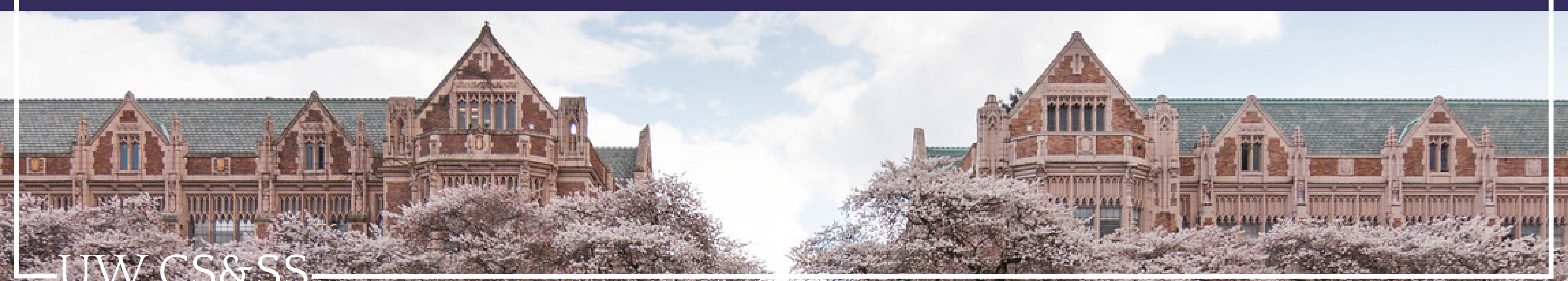
CSSS508, Week 7

Vectorization and Functions

Chuck Lanfear

Nov 18, 2020

Updated: Nov 24, 2020



UW CS&SS

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 Week

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
##         70.14255         50.65957         16.48936         10.97872
```

Vectorization Avoids Loops

Loops are very powerful and applicable in almost any situation.

They are also often slower 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 actions you might perform often, 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
- Defining a new logical operator

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:

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

Test it out:

```
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: 0x000001ff87606bf0>
```

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 `my_new_func <- 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:

```
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%      Top 
## -2.333593597 -1.642744909 -1.291610989 -0.672982453 -0.006031143  0.660022100  1.24356
##       Top 5%       Top 1% 
##  1.607643031  2.325196350
```

An Aside on Apply functions

Don't Loop, `apply()` Yourself Instead

Writing loops is challenging, particularly for new coders.

Loops also require writing a lot of code and are hard to troubleshoot.

But loops aren't the only way to iterate in R.

Like a loop, `apply` functions iterate over elements of objects, except:

- They don't need preallocation--you can directly assign the output.
- They *must use a function*

Nearly anything you can do with an explicit loop can be done more easily with the `apply` family of 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%    Top 10%     Top 5%     Top 1%
##     38.588     47.580     56.240     64.700     70.400     78.450     84.600     90.670     92.454
##
## $Agriculture
##  Bottom 1%  Bottom 5% Bottom 10% Bottom 25%     Median    Top 25%    Top 10%     Top 5%     Top 1%
##      4.190     15.650     17.360     35.900     54.100     67.650     76.820     84.810     87.952
##
## $Examination
##  Bottom 1%  Bottom 5% Bottom 10% Bottom 25%     Median    Top 25%    Top 10%     Top 5%     Top 1%
##       3.00       5.00       6.00      12.00      16.00      22.00      26.00      30.40      36.08
```

`sapply()`: Simple `lapply()`

A downside to `lapply()` is that lists are hard to work with. `sapply()` **s**implifies the output by making each element a column in a matrix… usually:

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

```
##            Fertility Agriculture Examination Education Catholic Infant.Mortality
## Bottom 1%     38.588       4.190        3.00      1.46   2.2052           12.778
## Bottom 5%     47.580      15.650        5.00      2.00   2.4480           15.600
## Bottom 10%    56.240      17.360        6.00      3.00   2.8320           16.420
## Bottom 25%    64.700      35.900       12.00      6.00   5.1950           18.150
## Median        70.400      54.100       16.00      8.00  15.1400           20.000
## Top 25%       78.450      67.650       22.00     12.00  93.1250           21.700
## Top 10%       84.600      76.820       26.00     23.20  99.0000           23.680
## Top 5%        90.670      84.810       30.40     29.00  99.6140           24.470
## Top 1%        92.454      87.952       36.08     43.34  99.8666           25.818
```

`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`).

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

```
##             Fertility Agriculture Examination Education Catholic Infant.Mortality
## Bottom 1%      38.588       4.190        3.00      1.46   2.2052           12.778
## Bottom 5%      47.580      15.650        5.00      2.00   2.4480           15.600
## Bottom 10%     56.240      17.360        6.00      3.00   2.8320           16.420
## Bottom 25%     64.700      35.900       12.00      6.00   5.1950           18.150
## Median         70.400      54.100       16.00      8.00  15.1400           20.000
## Top 25%        78.450      67.650       22.00     12.00  93.1250           21.700
## Top 10%        84.600      76.820       26.00     23.20  99.0000           23.680
## Top 5%         90.670      84.810       30.40     29.00  99.6140           24.470
## Top 1%         92.454      87.952       36.08     43.34  99.8666           25.818
```

Data Loading with Loop

Remember the loop for loading data files from last week?

```
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
```

Data Loading with `lapply()`

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

```
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
```

Data Loading with `vroom`

The fastest and easiest way is to use a fully vectorized data loading function, like `vroom::vroom()`!

```
library(vroom)
complete_data <- vroom(file_paths)
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
```

Just give `vroom()` a vector of file locations and it determines their delimiter, loads them all (crazy fast), and binds them into one dataframe.

From Loop to `apply()`

Converting code in a loop to an `apply` function is straightforward:

1. What you iterate over in the loop (e.g. `seq_along(x)`) becomes the first input.
2. The body of the loop becomes a function.
    - This function should take only the iterator index (e.g. `i`) as an input.
3. Assign the output to what your loop stored values in.

Loop vs. Apply

```r
loop_vec <- numeric(5)           # Preallocation!
for(x in seq_along(loop_vec)){ # Change x to 1,2,3,4,5
  loop_vec[x] <- x^2             # Write x squared to loop_vec
}
loop_vec
```

```
## [1]  1  4  9 16 25
```

`seq_along(loop_vec)` is just `1:5`, but we need the empty `loop_vec` to store results.

```r
# No preallocation, just iterate over 1:5 and assign output!
apply_vec <- sapply(1:5, function(x){x^2})
apply_vec
```

```
## [1]  1  4  9 16 25
```

For apply functions, we don't need to prellocate, so we just `sapply()` over `1:5` directly.

Back to Making and Using 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

Bucketing Function

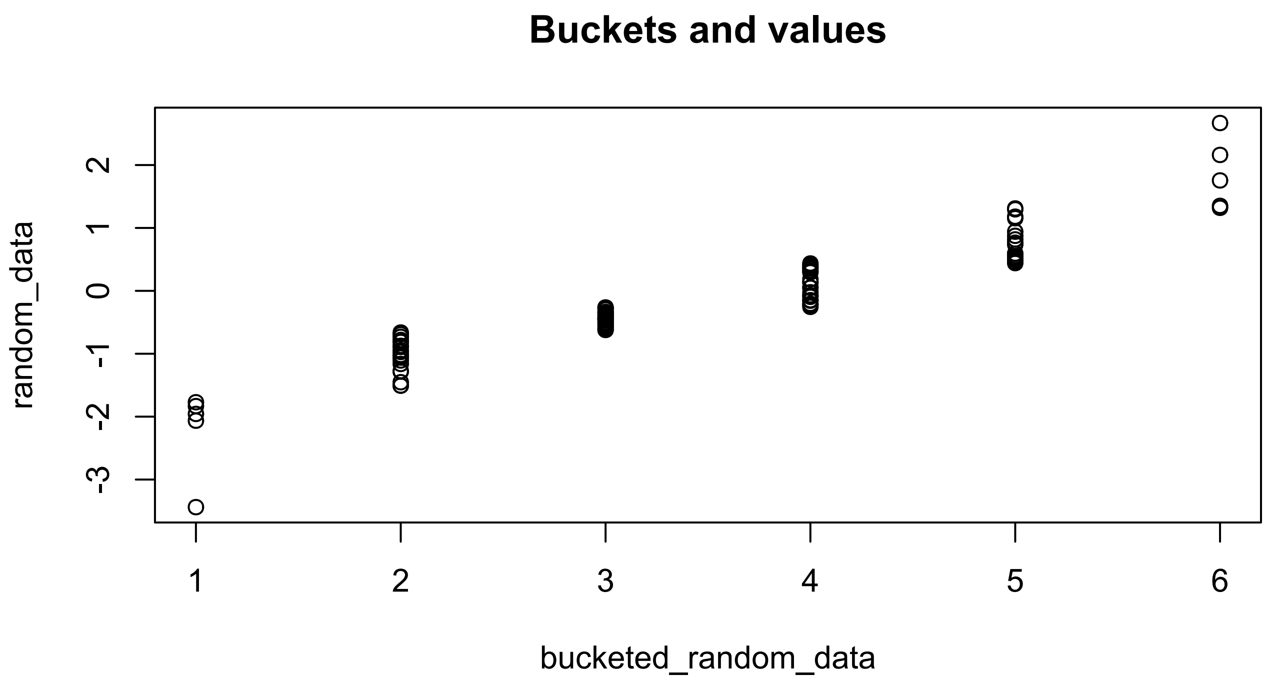
There's already a function in R called `cut()` that does this, but you need to tell it cutpoints or the number of buckets.

Let's make a convenience function that calls `cut()` using quantiles for splitting and returns an integer:

```
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))
}
```

Trying Out `bucket()`

```
random_data <- rnorm(100)
bucketed_random_data <- bucket(random_data,
                           quants = c(0.05, 0.25, 0.5, 0.75, 0.95))
plot(x = bucketed_random_data, y = random_data, main = "Buckets and values")
```



Impossible Values

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

```
(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
```

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()`

The `dplyr` function `across()` allows us to a function to every variable (besides `school`) to update the columns in `school_data`:

```r
library(dplyr)
school_data %>%
   mutate(across(-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
```

Standard and 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 programming with `dplyr` easier.

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

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

Anonymous Functions in `dplyr`

You can skip naming your function in `dplyr` if you won't use it again. Code below will return the mean divided by the standard deviation for each variable in `swiss`:

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

```
##   Fertility Agriculture Examination Education  Catholic Infant.Mortality
## 1  5.615134    2.230597    2.066884  1.141785 0.9865478         6.846766
```

Anonymous `lapply()`

Like with `dplyr`, you can use anonymous functions in `lapply()`, 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] 5.615134
##
## $Agriculture
## [1] 2.230597
##
## $Examination
## [1] 2.066884
##
## $Education
## [1] 1.141785
##
## $Catholic
## [1] 0.9865478
```

Extended Example:

`ggplot2` Templates

Flexible `ggplot2`

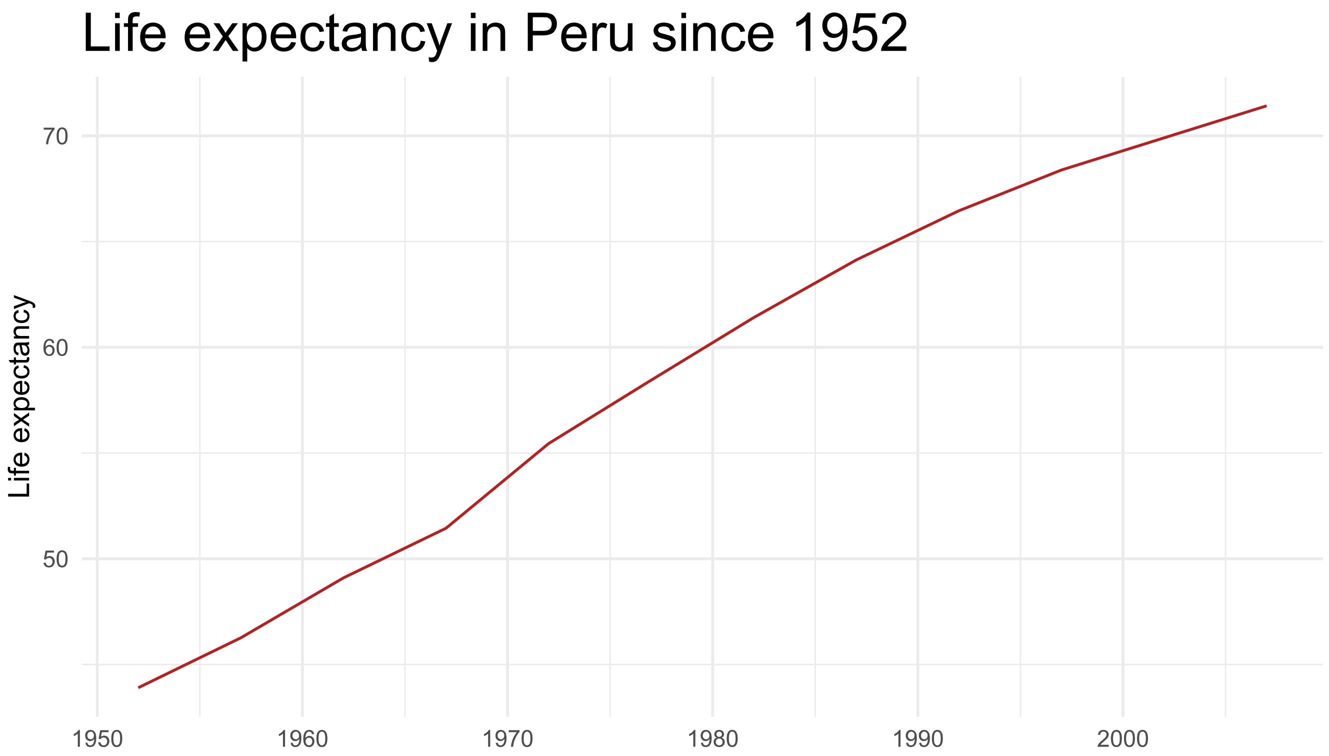
Let's say you have a particular way you like your charts:

```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

Another Example



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:

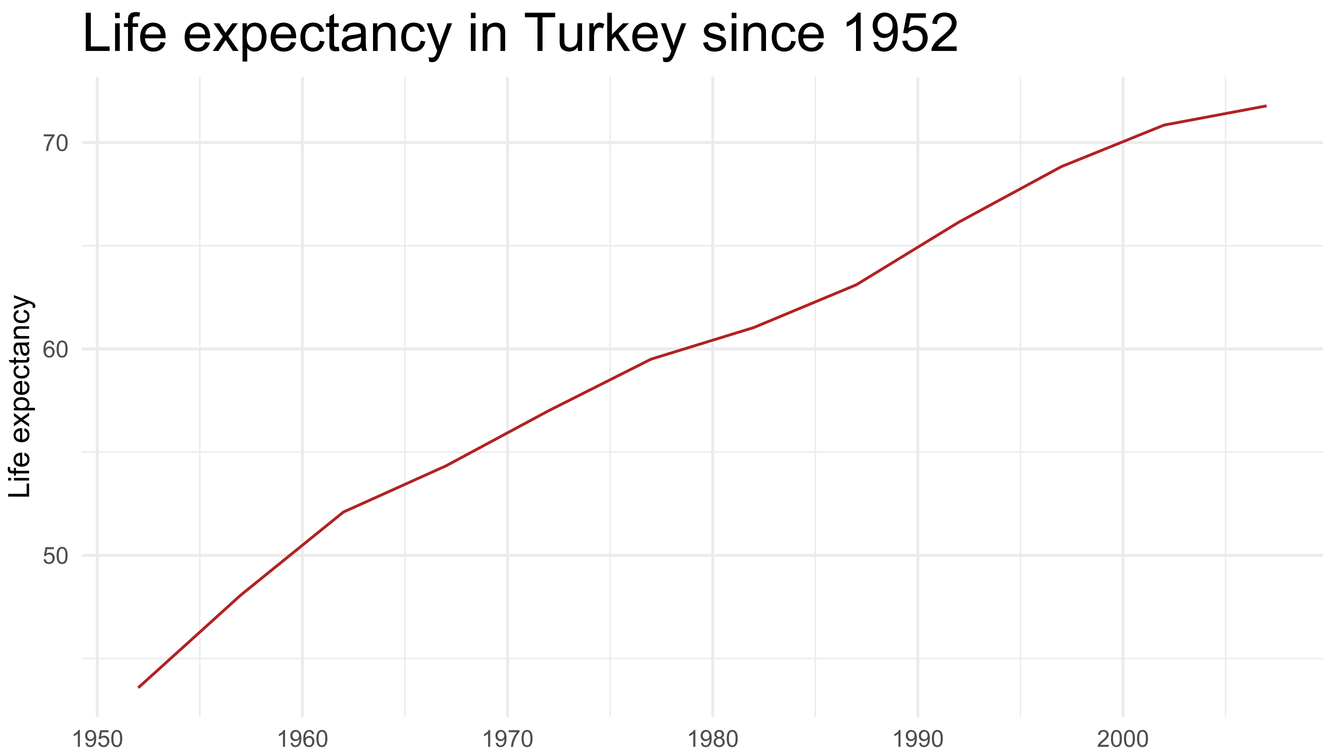
```
gapminder_lifeplot <- function(cntry) {
    ggplot(gapminder %>% filter(country == cntry),
        aes(x = year, y = lifeExp)) +
    geom_line(color = "firebrick") +
    xlab(NULL) + ylab("Life expectancy") + theme_minimal() +
    ggtitle(paste0("Life expectancy in ", cntry, " since 1952")) +
    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")
```



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:

```
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)"
```

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

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

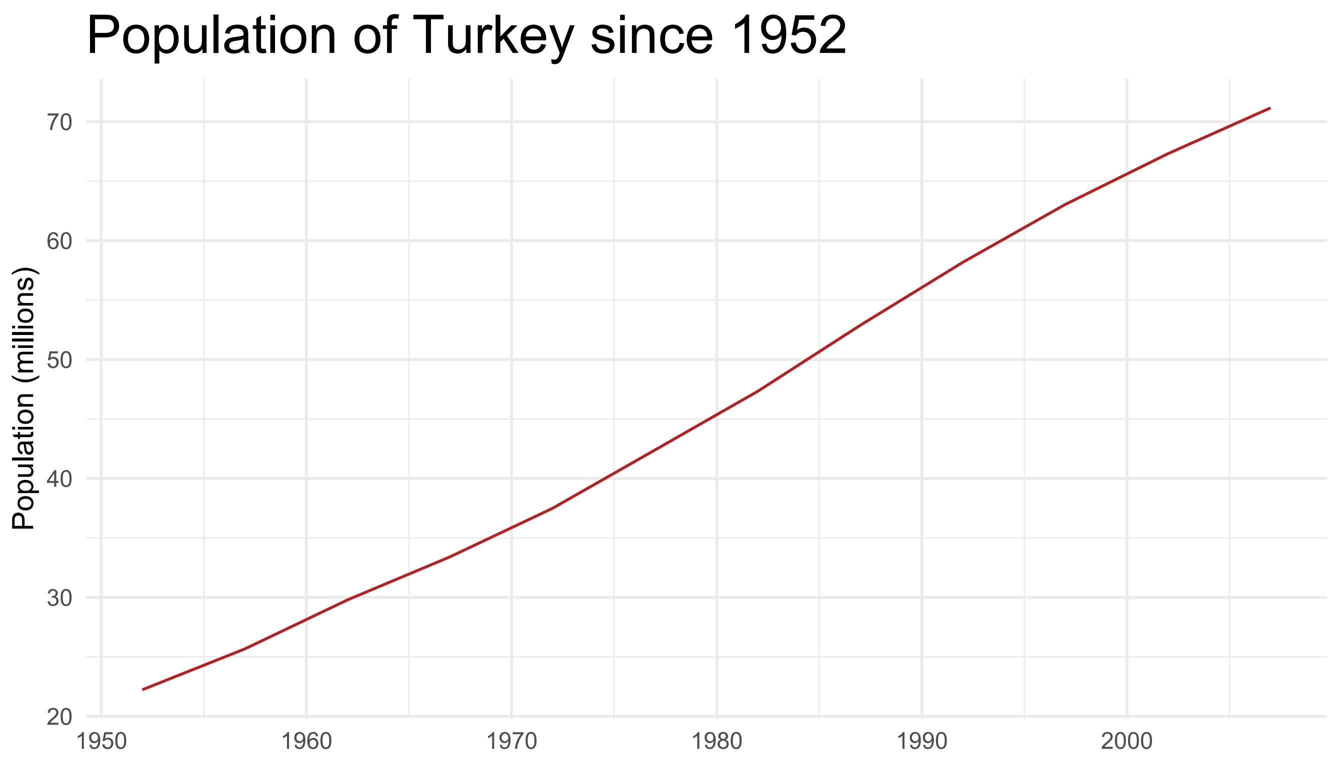
`aes_string()`

`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:

```
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")
```



Making an Operator

Opposite of `%in%`

`%in%` returns `TRUE` where elements on its left equal any element on the right.

```
us_ca <- c("Canada", "United States")
gapminder %>% filter(country %in% us_ca) %>% distinct(country) %>% head(2)
```

```
## # A tibble: 2 x 1
##   country
##   <fct>
## 1 Canada
## 2 United States
```

We can invert this to get the opposite, but it looks a bit awkward:

```
gapminder %>% filter(!country %in% us_ca) %>% distinct(country) %>% head(2)
```

```
## # A tibble: 2 x 1
##   country
##   <fct>
## 1 Afghanistan
## 2 Albania
```

`%!in%`

We can *invert* or **negate**[1] `%in%` to get a "not in" operator:

```
`%!in%` <- Negate(`%in%`)
```

To make a new operator, you need to put it in backticks.

```
gapminder %>%
  filter(country %!in% us_ca) %>% # Our new operator!
  distinct(country) %>%
  head(2)
```

```
## # A tibble: 2 x 1
##   country
##   <fct>
## 1 Afghanistan
## 2 Albania
```

[1] `Negate()` produces logical negations of *functions*, inverting their output. e.g.: `isnt.numeric <- Negate(is.numeric)`

Wrapping Up

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*!

Bonus Function

My lectures are rendered with a function!

```r
render_and_print_slides <- function(week){
    week_dir     <- paste0(getwd(), "/Lectures/", "Week", week, "/")
    current_rmd  <- paste0(week_dir, stringr::str_subset(list.files(week_dir),
                                                      "^CSSS508_Week.*\\.Rmd$"))
    rmarkdown::render(current_rmd, encoding = "UTF-8")
    current_html <- stringr::str_replace(current_rmd,  "\\.Rmd",  "\\.html")
    new_pdf_file <- stringr::str_replace(current_html, "\\.html", "\\.pdf")
    new_r_script <- stringr::str_replace(current_html, "\\.html", "\\.R")
    message("Slides rendered, waiting 5 seconds.")
    Sys.sleep(5)
    message("Purling slides.")
    knitr::purl(input = current_rmd, output = new_r_script, documentation = 0)
    message("Printing from Chrome.")
    pagedown::chrome_print(current_html, format="pdf")
    message(paste0("Printing complete at ", week_dir))
}
```

I give it a numeric week and it (1) finds the lecture `.Rmd`, (2) knits the slides, (3) creates a `.R` file, (4) then opens the slides in Chrome and prints a PDF.

Homework

Download and analyze data from the first year of Seattle's Pronto! bike sharing program.

Using the provided template, you will write:

1. A loop (or `lapply()`) to read in the data from multiple files.
    - Don't just use `vroom()`!
2. Functions to clean up the data
3. A function to visualize ridership over the first year.

There is some string processing needed—much of which you have already seen or can probably Google—but *some will come in the next lecture*. I give suggestions in the template, but I can cover string processing in detail in lab if needed before the homework is due.

PART 1 DUE: Next week

PART 2 DUE: In two weeks