SWC R Workshop @ USDA: Notes and Code

Preethy Nair

July 23, 2020

Contents

Chapter 1

Information

This is a supplementary notebook for the online SWC R workshop at the USDA on July 23, 2020. The notebook includes the code that was used during workshop and notes based on the Carpentries R lesson, R for Reproducbile data analyis.

Please see the course website to access other lessons we covered for the workshop. The workshop notes for other sessions are provided in the collaborative document.

Summary of yesterday's `R` lessons are also provided in the same collaborative note.

To get help for a function in `R`, we can use `help`.

```
help("%in%") # value matching
help("c")  # combine function
help("%>%") # pipe operator
help("<-") # assignment operator
help("-") # arithmetic operator
```

1.1 Prerequisites

To participate in this workshop session on **R**, you need to install **R**, **RStudio** and some packages from `R` (R Core Team, 2019) (`dplyr` (Wickham et al., 2020), `tidyr`(Wickham and Henry, 2020), `knitr` (Xie, 2019, 2015)). Please see the course website for instructions to install `R` and `RStudio`. Once you have **R** installed on your computers, the packages required for this session can be installed using the below commands.

```r
install.packages(c("dplyr", "tidyr", "knitr"))
# or individually
install.packages("dplyr")
install.packages("tidyr")
install.packages("knitr")
```

Chapter 2

Vectorization

Most of R's functions are vectorized.

- These functions will operate on all elements of a vector at once
- No need to loop through and act on each element one at a time

Advantages of vectorisation We can write code that is:

- concise
- easy to read and
- less error prone.

e.g.

```
x <- 1:4
x
```

```
## [1] 1 2 3 4
```

```
x * 2
```

```
## [1] 2 4 6 8
```

Above, each element of the vector was multiplied by 2.
Lets see how vectorisation works while adding two vectors in `R`.

```
y <- 6:9
y
```

```
## [1] 6 7 8 9
```

```
x + y
```

```
## [1]  7  9 11 13
```

Here, each element of `x` was added to its corresponding element of `y`. Lets see that again:

```
x; y; x+y
```

```
## [1] 1 2 3 4
```

```
## [1] 6 7 8 9
```

```
## [1]  7  9 11 13
```

Another way to implement vectorisation while adding two vectors together is by using a `for` loop.

2.1 Structure of a `for()` loop

```
for (iterator in set_of_values) {
          do a thing
     }
```

- indentation is not important in R compared to Shell and Python
- but its a good practice to use indentation

Uses of for() loop

- repeating operations

Example of a for loop

```
for (i in 1:10) {
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

`for()` loop is good for iterating over a set of values, when the order of iteration is important. We saw `for()` loop yesterday.

Its good to learn for() loops, but avoid using for() loops unless the order of iteration is important.

Here's an example of a for loop that does vector addition.

```
x
```

```
## [1] 1 2 3 4
```

```
y
```

```
## [1] 6 7 8 9
```

```
#vectorisation using for loop
output_vector <- c()
for (i in 1:4) {
    output_vector[i] <- x[i] + y[i]
}
output_vector
```

```
## [1]  7  9 11 13
```

Compare this to the output using vectorised operations.

```
x
```

```
## [1] 1 2 3 4
```

```
y
```

```
## [1] 6 7 8 9
```

```
sum_xy <- x + y
sum_xy
```

```
## [1]  7  9 11 13
```

```
identical(output_vector, sum_xy)
```

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

Both are identical.

Let's try Vectorisation in gapminder data. Let's load the gapminder data that we used yesterday in the first part of our R lesson.

You can clear your environment for a fresh start by removing all the objects using the below command.

```
rm(list = ls())
```

Please import the `gapminder` data to your `RStudio` environment. If you do not have the `gapminder` loaded into your RStudio session, you can do it using the below code chunk. This code chunk is also provided in the collaborative document.

```
# loading data using a link
gapminder <- read.csv("https://raw.githubusercontent.com/swcarpentry/r-novice-gapminder/gh-pages/
```

```
# if the gapminder is already in your data folder
# you can load it using the below code:
#gapminder <- read.csv("data/gapminder_data.csv", stringsAsFactors=FALSE)
```

Examine the gapminder dataset.

```
head(gapminder)  # shows the first few rows of the dataset
```

```
##       country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134
```

```
tail(gapminder)  # shows the last few rows of the dataset
```

```
##       country year      pop continent lifeExp gdpPercap
## 1699 Zimbabwe 1982  7636524    Africa  60.363  788.8550
## 1700 Zimbabwe 1987  9216418    Africa  62.351  706.1573
## 1701 Zimbabwe 1992 10704340    Africa  60.377  693.4208
## 1702 Zimbabwe 1997 11404948    Africa  46.809  792.4500
## 1703 Zimbabwe 2002 11926563    Africa  39.989  672.0386
## 1704 Zimbabwe 2007 12311143    Africa  43.487  469.7093
```

Lets try vectorisation on the `pop` column of the `gapminder` dataset.

1. Make a new column in the `gapminder` data frame that contains population in units of millions of people.
2. Check the head or tail of the data frame to make sure it worked.

```
gapminder$pop_millions <- gapminder["pop"] / 1e6
#or
gapminder$pop_millions_2 <- gapminder$pop / 1e6
# or
gapminder["pop_millions_3"] <- gapminder$pop / 1e6
```

Check the head or tail of the data frame to make sure it worked.

```
head(gapminder)
```

```
##       country year      pop continent lifeExp gdpPercap       pop
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453  8.425333
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530  9.240934
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007 10.267083
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971 11.537966
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811 13.079460
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134 14.880372
```

```
##   pop_millions_2 pop_millions_3
## 1       8.425333       8.425333
## 2       9.240934       9.240934
## 3      10.267083      10.267083
## 4      11.537966      11.537966
## 5      13.079460      13.079460
## 6      14.880372      14.880372
```

```
# check if both the created variables are identical or not
# using the R function identical()
identical(gapminder$pop_millions, gapminder$pop_millions_2 )
```

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

In addition to mathematical operators, Comparison operators, logical operators & many functions are also vectorized:

Let's see the **Comparison operators**

```
x <- 0:6
x
```

```
## [1] 0 1 2 3 4 5 6
```

```
x > 2
```

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

Logical operators

```
a <- x > 3  # or, for clarity, a <- (x > 3)
a
```

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

Functions `log()`

```
x
```

```
## [1] 0 1 2 3 4 5 6
```

```
x <- 1:4
x
```

```
## [1] 1 2 3 4
```

```
log(x)
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944
```

Vectorisation in matrices Vectorized operations work element-wise on matrices.

```
m <- matrix(1:12, nrow=3, ncol=4)
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
m * -1 # element wise operation
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   -1   -4   -7  -10
## [2,]   -2   -5   -8  -11
## [3,]   -3   -6   -9  -12
```

```
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
m ^ -1 # element wise operation
```

```
##           [,1]      [,2]      [,3]       [,4]
## [1,] 1.0000000 0.2500000 0.1428571 0.10000000
## [2,] 0.5000000 0.2000000 0.1250000 0.09090909
## [3,] 0.3333333 0.1666667 0.1111111 0.08333333
```

Very important:

- the operator $* => $ element-wise multiplication
- for matrix multiplication, use the %*% operator

```
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
matrix(1, nrow=4, ncol=1)
```

```
##      [,1]
## [1,]    1
## [2,]    1
## [3,]    1
## [4,]    1
```

```
m %*% matrix(1, nrow=4, ncol=1)
```

```
##      [,1]
## [1,]   22
## [2,]   26
## [3,]   30
```

```r
matrix(1:4, nrow=1) #row matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
```

```r
matrix(1:4, nrow=1) %*% matrix(1:4, ncol=1)
```

```
##      [,1]
## [1,]   30
```

Chapter 3

Functions

Functions

- are the basic building block of most programming languages
- are used to gather a sequence of operations into a whole, preserving it for ongoing use.
- have names that we can remember and call
- have a defined set of inputs and expected outputs
- provide connections to the larger programming environment

We can repeat several operations with a single function call.

There are 2 types of functions:

1. built-in functions
2. user-defined functions

Below are some of the built-in functions from **base R**.

```
log()
mean()
min()
sd()
```

User-defined functions are created in **R** using the keyword **function**. See below regarding how to get help for writing functions in **R**.

```
?"function"
```

3.1 Components of a function

Below presented are the components of a function:

a. name
b. arguments/inputs - within parentheses -()
c. body - within curly braces - {}
    - statements that are executed when it runs

The statements in the body are indented by two spaces as a good practice for writing `R` code. A function is called in `R` by providing the function name with the arguments in parenthesis, like shown below.

```
# defining a function
function_name <- function(arguments){
    body
}

# this is how we call or use a function
function_name(arguments)
```

Let's define a function `fahr_to_kelvin()` that converts temperatures from Fahrenheit to Kelvin:

```
fahr_to_kelvin <- function(temperature) {
    kelvin <- ((temperature - 32) * (5 / 9)) + 273.15
    return(kelvin)
}
```

Run the above code chunk on your RStudio console.

Next, check if you have executed the code chunk by going through the `History` tab of the top right panel on your RStudio window. You can also make sure that the function named `fahr_to_kelvin` is available for you to use by searching the **RStudio** `Environment`. From the top right panel of your RStudio window, go the tab `Environment` and search for the function `fahr_to_kelvin` under the Function names.

Lets try using our function to convert the freezing point of water from Fahrenheit to kelvin. Freezing point of water is 0 °C or 32 °F.

```
fahr_to_kelvin(32)
```

```
## [1] 273.15
```

Next, use the same function to convert the boiling point of water from Fahrenheit to Kelvin. Note that the boiling point of water is 100 °C (212 °F).

```
fahr_to_kelvin(212)
```

```
## [1] 373.15
```

3.2 Challenge 1

Write a function called kelvin_to_celsius() that takes a temperature in Kelvin & converts that to temperature in Celsius.

To convert from Kelvin to Celsius you subtract 273.15

Reference: http://swcarpentry.github.io/r-novice-gapminder/10-functions/index.html

```
kelvin_to_celsius <- function(temp) {
    .....
    return(celsius)
}
```

Solution

```
kelvin_to_celsius <- function(temp) {
    celsius <- temp - 273.15
    return(celsius)
}
```

3.3 Combining functions

We have already defined two functions:

1. `fahr_to_kelvin()` - converts from Fahrenheit to Kelvin
2. `kelvin_to_celsius()` - converts from Kelvin to Celsius

See if they are available in your global environment of RStudio.

The real power of functions comes from mixing, matching and combining them into ever-larger chunks to get the effect we want.

Lets try to understand how to combine functions.

We will define a third function to convert directly from Fahrenheit to Celsius, by reusing the two functions above. Let the name of third function be `fahr_to_celsius()`

- input: temperature in Fahrenheit
- expected output: temperature in Celsius

We are going to accomplish this in 3 steps:

1. Convert Fahrenheit to Kelvin : `fahr_to_kelvin()`
2. Convert Kelvin from Step1 to Celsius: `kelvin_to_celsius()`
3. Return the above result at the end of the function

```
fahr_to_celsius <- function(temp) {
    temp_k <- fahr_to_kelvin(temp)
    result <- kelvin_to_celsius(temp_k)
    return(result)
}

# or can nest functions like this
fahr_to_celsius_2 <- function(temp) {
    temp_c <- kelvin_to_celsius(fahr_to_kelvin(temp))
    return(temp_c)
}
```

Lets try to convert the boiling and freezing point of water from Fahrenheit to Celsius. First, make sure all the three functions are available in your global environment.

Call your combined function for the above mentioned conversions as shown below

```
fahr_to_celsius(212) # boiling point of water
```

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

```
fahr_to_celsius(32) # freezing point of water
```

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

3.4 Defensive Programming

- Checking function parameters
- writing functions provides an efficient way to make R code re-usable and modular
- it is important to ensure that functions only work in their intended use-cases.
- frequently check function parameters using conditions and throw an error if something is wrong.
- These checks are referred to as assertion statements because we want to assert some condition is TRUE before proceeding.
- make it easier to debug because they give us a better idea of where the errors originate.
- How to use assertion in `R` ?
  - we can tell `R` to stop executing the function using stop()

3.4.1 Case 1: Functions with one argument

We can create conditions with `stopifnot()`. Let's start by re-examining `fahr_to_kelvin()`.

- Here, we know that the argument `temp` must be a `numeric`
  - Otherwise, the mathematical operations used in the temperature conversions will not work.
- Hence, we check this with an `if` statement and `stop()` if the condition is violated

```
is.numeric("TRUE")
```

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

```
is.numeric(TRUE)
```

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

```
is.numeric(2)
```

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

```
fahr_to_kelvin <- function(temp) {
    if (!is.numeric(temp)) {
        stop("temp must be a numeric vector.")
    }
    kelvin <- ((temp - 32) * (5 / 9)) + 273.15
    return(kelvin)
}
```

3.4.2 Case 2: Multiple arguments or conditions

- Use `stopifnot()` for type-assertion
- works in the same way as `stop()`
- shorter syntax
- throws an error if it finds a condition that is `FALSE`.

Here is the code for defensive programming with `stopifnot()`

```
fahr_to_kelvin_stopifnot <- function(temp) {
    stopifnot(is.numeric(temp))
    kelvin <- ((temp - 32) * (5 / 9)) + 273.15
    return(kelvin)
}
```

Note - We can list as many conditions that should evaluate to `TRUE` - Secondary purpose: extra documentation for the function.

Lets try checking the impact of the newly added assertions on our function `fahr_to_kelvin()`.

- Check the Global environment

If you type your function name without parenthesis, R will print out the code for your function.

```
fahr_to_kelvin_stopifnot
```

```
## function(temp) {
##     stopifnot(is.numeric(temp))
##     kelvin <- ((temp - 32) * (5 / 9)) + 273.15
##     return(kelvin)
## }
```

Freezing point of water

```
fahr_to_kelvin(temp = 32)
```

```
## [1] 273.15
```

```
fahr_to_kelvin_stopifnot(temp = 32)
```

```
## [1] 273.15
```

Let's convert the number 32 to the type `factor` using the function `as.factor()`. This is an example of a type coercion that we saw on day 1 of the R lesson.

```
as.factor(32)
```

```
## [1] 32
## Levels: 32
```

The metric is a factor instead of numeric

```
fahr_to_kelvin(temp = as.factor(32))
fahr_to_kelvin_stopifnot(temp = as.factor(32))
```

Both functions still works when given proper input. But fails instantly if given improper input.

We will define a function `calcGDP()` that calculates the Gross Domestic Product of a nation from the `gapminder` dataset that we used in the chapter `Vectorisation`.

`calcGDP` takes a dataset and multiplies the population column `pop` with the GDP per capita column `gdpPercap` to get the GDP.

```
head(gapminder)
```

```
##       country year      pop continent lifeExp gdpPercap       pop
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453  8.425333
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530  9.240934
```

```
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007 10.267083
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971 11.537966
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811 13.079460
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134 14.880372
##   pop_millions_2 pop_millions_3
## 1       8.425333       8.425333
## 2       9.240934       9.240934
## 3      10.267083      10.267083
## 4      11.537966      11.537966
## 5      13.079460      13.079460
## 6      14.880372      14.880372
```

```
calcGDP <- function(dat) {
    gdp <- dat$pop * dat$gdpPercap
    return(gdp)
}
```

Lets see what this function returns when applied to a part of the gapminder data.

```
calcGDP(head(gapminder))
```

```
## [1]  6567086330  7585448670  8758855797  9648014150  9678553274 11697659231
```

As we want our results to be more informative than the above, let's add some more arguments so we can extract per year and country GDP. We can also apply 1. defensive programming & 2. default values for the arguments

```
calcGDP <- function(dat, year=NULL, country=NULL) {
    if(!is.null(year)) {
        dat <- dat[dat$year %in% year, ]
    }
    if (!is.null(country)) {
        dat <- dat[dat$country %in% country,]
    }
    gdp <- dat$pop * dat$gdpPercap

    new <- cbind(dat, gdp=gdp)
    # modified gapminder with a gdp column
    return(new)
}
```

Explanation of what the function `calcGDP` does:

i) subsets the provided data by year if the year argument isn't empty
ii) subsets the result by country if the country argument isn't empty.
iii) calculates the GDP for the resultant subset
iv) adds the GDP as a new column to the subsetted data
v) returns the subsetted data (if year or/and country was present in the input

dataframe) with a gdp column added as the final result.

Note: Implementing defensive programming can make our functions more flexible for later use.

We can use it to calculate the GDP for:

- The whole dataset;
- A single year;
- A single country;
- A single combination of year and country.

By using %in% instead, we can also give multiple years or countries to those arguments. Please see help for **%in%** which we covered yesterday using the below functions:

```
help("%in%")
?"%in%"
```

Let's take a look at what happens when we specify the year:

```
head(calcGDP(gapminder, year = 2007))
```

```
##        country year      pop continent lifeExp  gdpPercap       pop
## 12 Afghanistan 2007 31889923      Asia  43.828   974.5803 31.889923
## 24     Albania 2007  3600523    Europe  76.423  5937.0295  3.600523
## 36     Algeria 2007 33333216    Africa  72.301  6223.3675 33.333216
## 48      Angola 2007 12420476    Africa  42.731  4797.2313 12.420476
## 60   Argentina 2007 40301927  Americas  75.320 12779.3796 40.301927
## 72   Australia 2007 20434176   Oceania  81.235 34435.3674 20.434176
##    pop_millions_2 pop_millions_3          gdp
## 12      31.889923      31.889923  31079291949
## 24       3.600523       3.600523  21376411360
## 36      33.333216      33.333216 207444851958
## 48      12.420476      12.420476  59583895818
## 60      40.301927      40.301927 515033625357
## 72      20.434176      20.434176 703658358894
```

Or for a specific country:

```
calcGDP(gapminder, country="Australia")
```

```
##      country year      pop continent lifeExp gdpPercap       pop
## 61 Australia 1952  8691212   Oceania  69.120  10039.60  8.691212
## 62 Australia 1957  9712569   Oceania  70.330  10949.65  9.712569
## 63 Australia 1962 10794968   Oceania  70.930  12217.23 10.794968
## 64 Australia 1967 11872264   Oceania  71.100  14526.12 11.872264
## 65 Australia 1972 13177000   Oceania  71.930  16788.63 13.177000
## 66 Australia 1977 14074100   Oceania  73.490  18334.20 14.074100
## 67 Australia 1982 15184200   Oceania  74.740  19477.01 15.184200
```

```
## 68 Australia 1987 16257249   Oceania  76.320  21888.89 16.257249
## 69 Australia 1992 17481977   Oceania  77.560  23424.77 17.481977
## 70 Australia 1997 18565243   Oceania  78.830  26997.94 18.565243
## 71 Australia 2002 19546792   Oceania  80.370  30687.75 19.546792
## 72 Australia 2007 20434176   Oceania  81.235  34435.37 20.434176
##    pop_millions_2 pop_millions_3          gdp
## 61       8.691212       8.691212  87256254102
## 62       9.712569       9.712569 106349227169
## 63      10.794968      10.794968 131884573002
## 64      11.872264      11.872264 172457986742
## 65      13.177000      13.177000 221223770658
## 66      14.074100      14.074100 258037329175
## 67      15.184200      15.184200 295742804309
## 68      16.257249      16.257249 355853119294
## 69      17.481977      17.481977 409511234952
## 70      18.565243      18.565243 501223252921
## 71      19.546792      19.546792 599847158654
## 72      20.434176      20.434176 703658358894
```

Let's do Challenges 4 & 5

Reference: http://swcarpentry.github.io/r-novice-gapminder/10-functions/index.html

3.5 Challenge 4

Test out your GDP function by calculating the GDP for New Zealand in 1987. How does this differ from New Zealand's GDP in 1952?

```
calcGDP(gapminder, year = ..., country = ..)
```

```
# see if you get any results with these options. If you don't get results, can you tell why ?
calcGDP(gapminder, year = 1952, country = "Newzealand")
```

```
##  [1] country        year           pop            continent
##  [5] lifeExp        gdpPercap      pop_millions   pop_millions_2
##  [9] pop_millions_3 gdp
## <0 rows> (or 0-length row.names)
```

```
calcGDP(gapminder, year = 1952, country = "NewZealand")
```

```
##  [1] country        year           pop            continent
##  [5] lifeExp        gdpPercap      pop_millions   pop_millions_2
##  [9] pop_millions_3 gdp
## <0 rows> (or 0-length row.names)
```

```
calcGDP(gapminder, year = 1952, country = "New Zealand")
```

```
##          country year     pop continent lifeExp gdpPercap      pop
## 1093 New Zealand 1952 1994794   Oceania   69.39  10556.58 1.994794
##      pop_millions_2 pop_millions_3         gdp
## 1093       1.994794       1.994794 21058193787
```

3.6 Challenge 5

Write a function called fence() that takes two vectors as arguments, called text and wrapper, and prints out the text wrapped with the wrapper:

The paste() function, we learned yesterday, can be used to combine text together. See below.

```
best_practice <- c("Write", "programs", "for", "people", "not", "computers")
paste(best_practice, collapse=" ")
```

```
## [1] "Write programs for people not computers"
```

Let the argument `text` be `best_practice` and `wrapper` be `****`. The frame for the function is given below. You can try completing this.

```
fence <- function(text, wrapper){
    ...
    ....
    return(result)
}
# calling the function
fence(text=.., wrapper=..)
```

Solution

```
fence <- function(text, wrapper){
    text <- c(wrapper, text, wrapper)
    result <- paste(text, collapse = " ")
    return(result)
}

best_practice <- c("Write", "programs", "for", "people", "not", "computers")
fence(text=best_practice, wrapper="***")
```

```
## [1] "*** Write programs for people not computers ***"
```

3.7 How to use functions for later analyses

1. Save functions to a file: e.g.: "functions.R"
2. Use it in another R script by using the `source()` function
    - `source("~/functions/functions.R")`

3.8 Other important concepts

- Pass by value
- Function scope
- Testing and documenting

Chapter 4

Saving plots and Writing data

4.1 Saving plots in RStudio

i. `ggsave()`

```
library(ggplot2)
```

```
ggplot(data=gapminder,
       aes(x=year, y=lifeExp, colour=country)) +
    geom_line() +
    theme(legend.position = "none")
ggsave("PlotName.pdf")
```

ii. **Export from RStudio**

- Make your plot in the RStudio
- Go to the lower right panel
- Select the tab `Plots`
- Select the options for export from the drop-down menu
  - Save as image or Save as pdf or Copy to clipboard

iii. **Plotting devices**

Plotting devices can be used in the `R` and `RStudio` to save plots. We can control the size and resolution of the plots to be saved using arguments to the plotting devices. One such plotting device is `pdf`. The below code chunk can be used to save the `ggplot` as a `pdf` file with the specified height and width.

```
pdf("Life_Exp_vs_time.pdf", width=12, height=4)
ggplot(data=gapminder,
```

```
       aes(x=year, y=lifeExp, colour=country)) +
    geom_line() +
    theme(legend.position = "none")

# Make sure to turn off the pdf device.
# Else, the plot won't be saved.
dev.off()
```

`jpeg`, `png`, `tiff` etc. are other plotting devices available in `R` to save plots. They can be called with arguments similar to the `pdf()`.

```
jpeg(..)
png(..)
tiff(..)
```

4.2 Writing Table

We can use the `write.table()` function for writing tables, similar to `read.table()` we used for importing data.

We can select the gapminder data for Australia and write it to a table.

```
# create a directory named cleaned-data if its not there
if(!dir.exists("./cleaned-data")){
  dir.create("cleaned-data")
}
#subset the data
aust_subset <- gapminder[gapminder$country == "Australia",]
#write the data
write.table(aust_subset,
    file="cleaned-data/gapminder-aus.csv",
    sep=","
)
```

Lets use the **`Terminal`** tab on the lower left panel of the RStudio and check the data.

```
head cleaned-data/gapminder-aus.csv
```

For help on the `write.table` function, use the `help` function.

```
help(write.table)
```

Lets use additional arguments for the `write.table` function so that quotes are not present for the values (option `quotes=FALSE`) and row names are not written to the cleaned file.

```
write.table(
    gapminder[gapminder$country == "Australia",],
    file="cleaned-data/gapminder-aus.csv",
    sep=",", quote=FALSE, row.names=FALSE
)
```

Lets use the RStudio **Terminal** again and examine the cleaned file.

```
# note - this command is for using on the Terminal window
# and not on the RStudio console
head cleaned-data/gapminder-aus.csv
```

Chapter 5

dplyr

`dplyr` is one of the `R` libraries that implements functions for manipulating dataframes.

Advantages of using `dplyr` compared to base R

- functions (verbs) are easier to read
- reduce repetition
- reduce the probability of making errors
- saves typing
- **can combine functions using pipes (%>%)**

dplyr verbs:

- `select()`: select variables (columns); (subsetting columns)
- `filter()`: choose data based on values; (subsetting rows)
- `summarize()`: provide summary for the grouped dataframes
- `mutate()`: create new variables
- `arrange()`: order the rows
- `group_by()`: create grouped dataframes

Note: The order of operations is very important in `dplyr`

See the episode on `dplyr` from the SWC lesson for the code snippets that we used in the workshop. See also the `dplyr` documentation from the `tidyverse` website.

For more help on `dplyr`, use the vignette for the same from `RStudio` or `R` like shown below.

```
vignette("dplyr")
```

Chapter 6

Producing Reports With knitr

RMarkDown reference can be accessed from here.

```
This is a how a header for RMarkdown file looks like:

---
title: "RMarkdown demo"
author: "Author name"
date: "7/23/2020"
output: html_document
---
```

6.0.1 Adding Citation

The bibliography may have any of these formats.

This is how you do some inline code: $2 + 2 = 4$.

You can do subscripts (e.g., F_2) with F_2 and superscripts (e.g., F^2) with F^2.

If you know how to write equations in LaTeX, you can use $ $ and

to insert math equations, like $E = mc^2$

“‘

Table 6.1: Bibliography formats that can be used with RMarkdown

format	file.extension
CSL-JSON	.json
MODS	.mods
BibLaTeX	.bib
BibTeX	.bibtex
RIS	.ris
EndNote	.enl
EndNote XML	.xml
ISI	.wos
MEDLINE	.medline

Bibliography

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Wickham, H., François, R., Henry, L., and Müller, K. (2020). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.0.

Wickham, H. and Henry, L. (2020). *tidyr: Tidy Messy Data*. R package version 1.1.0.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2019). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.24.