Introduction to R for Disease Surveillance and Outbreak Forecasting: Day 1

First steps toward using R - basic data objects and functions

Michael Wimberly, Dawn Nekorchuk and Andrea Hess,
Department of Geography and Environmental Sustainability, University of Oklahoma

October 22 2018, Bahir Dar, Ethiopia

Contents

1 Introduction

This tutorial will provide an introduction to R, an open-source computer software system for designed for statistical computation and graphics. R includes an interpreted language combined with a run-time environment that provides scripting, graphics, and debugging capabilities. By learning how to use the R language and environment, you will gain access to an large collection of functions for data processing, data visualization, and statistical modeling that are freely available through add-on packages.

Today, we will start with the basics. You will first learn about the R *objects* that store various types of data. Then you will learn how to manipulate these objects using *functions* that typically take one or more objects as inputs, and then modify these inputs to produce a new object as the output. These concepts will be deminstrated by showing how R can be used to perform practical tasks, including making simple calculations, applying these calculations over larger datsets, querying subsets of data that meet one or more conditions, and carrying out standard statistical tests.

2 Working with R objects

2.1 Using R as a calculator

At the most basic level, R can be used as a simple calculator. You can type in just about any mathematical expression on the command line in the console, hit the Enter key, and R will give the answer. Alternately, you can type the mathematical expressions as lines in a script file and then Run one or more lines, which is the way that we will be presenting this demonstration. The output is printed to the console by default.

```
33
```

```
## [1] 33
```

```
23 + 46
```

```
## [1] 69
```

```
(160 + 13) * 2.1
```

```
## [1] 363.3
```

```
237.81 / (3.7^2 + sqrt(165.3)) - 26.23
```

```
## [1] -17.27189
```

In addition to just printing the results, we can save the outputs of mathematical expressions as variables using the left assignment operator `<-`. Creating variables allows us to store and reuse the information at a later time. R stores each variable as an *object*. It is helpful to choose object names that make it easy to remember the type of information that is stored.

```
x <- 15
x
```

```
## [1] 15
```

```
y <- 23 + 15 / 2
y
```

```
## [1] 30.5
```

```
z <- x + y
z
```

```
## [1] 45.5
```

```
case <- 238
population <- 34226
incidence <- 10000 * case/population
incidence
```

```
## [1] 69.53778
```

When we enter an object name on the command line or specify an object name on the line of a script, R will invoke the `print` method by default and output its value to the console. Note that the following two lines produce the same output.

```
x
```

```
## [1] 15
```

```
print(x)
```

```
## [1] 15
```

2.2 Vectors

When we work with real data, we typically need to keep track of multiple measurements of our variables. For example, we may have data on malaria cases at multiple health facilities and data collected over multiple weeks, months, and years. Therefore, we usually work with vectors, which are objects that can contain multiple values. For example, consider that we have malaria case data for twelve months of the year. We can use the `c()` (combine) function to assign these data to a vector object.

```
cases <- c(43, 56, 23, 67, 81, 150, 110, 122, 161, 238, 138, 74)
cases
```

```
##  [1]  43  56  23  67  81 150 110 122 161 238 138  74
```

In R, there are several handy functions that can be used to create regular sequences of numbers or repeated values. In this example, we would like to have a a vector containing month numbers from 1 through 12, and another vector that repeats the same annual population value for each month. We will accomplish this task by calling some R functions and providing arguments that specify how to create the vectors.

```
month <- seq(from=1, to=12, by=1)
month
```

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

```
month2 <- 1:12
month2
```

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

```
pop <- rep(x=population, times=12)
pop
```

```
##  [1] 34226 34226 34226 34226 34226 34226 34226 34226 34226 34226 34226
## [12] 34226
```

One of the best features of R is that it allows us to to use vector arithmentic to carry out element-wise mathematical operations without having to write code for looping. Note that if we specify a a single value, it will automatically be repeated for the entire length of the vector. So the following two statements produce the same output.

```
cases * 2
```

```
##  [1]  86 112  46 134 162 300 220 244 322 476 276 148
```

```
cases2 <- c(22, 5, 13, 16, 22, 34, 24, 56, 431, 88, 67, 45)
totcases <- cases + cases2
totcases
```

```
##  [1]  65  61  36  83 103 184 134 178 592 326 205 119
```

```
inc <- 10000 * cases/pop
inc
```

```
##  [1] 12.563548 16.361830  6.720037 19.575761 23.666219 43.826331 32.139309
##  [8] 35.645416 47.040262 69.537778 40.320224 21.620990
```

Vectors can also be used in more complex mathematical expressions and supplied as arguments to functions.

```
mean(inc)
```

```
## [1] 30.75148
```

```
sum(cases)
```

```
## [1] 1263
```

```r
length(inc)
```

```
## [1] 12
```

```r
sum(inc) / length(inc)
```

```
## [1] 30.75148
```

```r
var(cases)
```

```
## [1] 3654.75
```

```r
sum((cases - mean(cases))^2) / (length(cases)-1)
```

```
## [1] 3654.75
```

Subsets of data can be selected by specifying index numbers within brackets. Positive numbers are used to include subsets, and negative numbers are used to exclude subsets.

```r
cases[5]
```

```
## [1] 81
```

```r
cases[c(1, 3, 8, 11)]
```

```
## [1]  43  23 122 138
```

```r
cases[9:12]
```

```
## [1] 161 238 138  74
```

```r
cases[-2]
```

```
##  [1]  43  23  67  81 150 110 122 161 238 138  74
```

Logcial vectors can be used to select subsets that meet a given criterion or to assign a new value to a subset

```r
inc > 10
```

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

```r
incgt10 <- inc > 10
class(incgt10)
```

```
## [1] "logical"
```

```r
inc[incgt10]
```

```
##  [1] 12.56355 16.36183 19.57576 23.66622 43.82633 32.13931 35.64542
##  [8] 47.04026 69.53778 40.32022 21.62099
```

```r
inc2 <- inc[inc > 10]
inc2
```

```
##  [1] 12.56355 16.36183 19.57576 23.66622 43.82633 32.13931 35.64542
##  [8] 47.04026 69.53778 40.32022 21.62099
```

In addition to numeric and logical vectors, we can also have vectors of character data such as month names and we can be subset our data using these names.

```r
mname <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
class(mname)
```

```
## [1] "character"
```

```r
names(inc) <-mname
inc[c("Jun", "Jul", "Aug")]
```

```
##      Jun      Jul      Aug
## 43.82633 32.13931 35.64542
```

There is also a special type of vector called a factor that is used for categorial data analysis. If we wanted to use the vector of woreda names in an analysis comparing statistica across the different woredas, we need to convert it into a factor.

```r
worname <- c("Mecha", "Dembecha", "Libokemkem", "Fogera", "South Achefer")
worfact <- factor(worname)
worname
```

```
## [1] "Mecha"         "Dembecha"      "Libokemkem"    "Fogera"
## [5] "South Achefer"
```

```r
class(worfact)
```

```
## [1] "factor"
```

```r
worfact
```

```
## [1] Mecha         Dembecha      Libokemkem    Fogera        South Achefer
## Levels: Dembecha Fogera Libokemkem Mecha South Achefer
```

In some cases, we may also want to convert numerical vectors to factors. Consider an experiment in which patients have either been given an experimental drug treatment or placed in a control group. The treatment category is represented in the data by integer codes, where 1 = treatment and 2 = control. The following code converts the numerical data into a factor.

```r
treat_num <- c(2, 1, 1, 2, 1, 1, 1, 2, 2, 2, 1, 1)
summary(treat_num)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   1.000   1.000   1.417   2.000   2.000
```

```r
treat_fac <- factor(treat_num, labels = c("treatment", "control"))
summary(treat_fac)
```

```
## treatment   control
##         7         5
```

2.3 Matrices and lists

Multiple vectors of the same length can be combined to create a matrix, which is a two-dimensional object with columns and rows. All of the values in a matrix must be the same data type (for example, numeric, text, or logical). We can use the `cbind()` function to combine the vectors as columns and the 'rbind()' function to combine the vectors as rows.

```r
mat1 <- cbind(month, cases, pop, inc)
mat1
```

```
##     month cases   pop       inc
## Jan     1    43 34226 12.563548
## Feb     2    56 34226 16.361830
## Mar     3    23 34226  6.720037
## Apr     4    67 34226 19.575761
## May     5    81 34226 23.666219
## Jun     6   150 34226 43.826331
```

```
## Jul     7   110 34226 32.139309
## Aug     8   122 34226 35.645416
## Sep     9   161 34226 47.040262
## Oct    10   238 34226 69.537778
## Nov    11   138 34226 40.320224
## Dec    12    74 34226 21.620990
```

```r
class(mat1)
```

```
## [1] "matrix"
```

```r
rownames(mat1)
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
```

```r
colnames(mat1)
```

```
## [1] "month" "cases" "pop"   "inc"
```

Matrix elements can be extracted via subscripting in [row, column] format. Leaving a subscript blank will return all columns or rows.

```r
dim(mat1)
```

```
## [1] 12  4
```

```r
mat1[1,1]
```

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

```r
mat1[,4]
```

```
##       Jan       Feb       Mar       Apr       May       Jun       Jul
## 12.563548 16.361830  6.720037 19.575761 23.666219 43.826331 32.139309
##       Aug       Sep       Oct       Nov       Dec
## 35.645416 47.040262 69.537778 40.320224 21.620990
```

```r
mat1[2,]
```

```
##       month       cases         pop         inc
##     2.00000    56.00000 34226.00000    16.36183
```

Lists are ordered collections of objects. They are created using the list function and can be subscripted by component number or component name. Lists differ from vectors and matrices in that they can contain a mixture of different data types.

```r
my_list <- list(mycases = cases, mypop = population)
my_list
```

```
## $mycases
##  [1]  43  56  23  67  81 150 110 122 161 238 138  74
##
## $mypop
## [1] 34226
```

```r
my_list[[1]]
```

```
##  [1]  43  56  23  67  81 150 110 122 161 238 138  74
```

```r
my_list[[2]]
```

```
## [1] 34226
```

```
my_list$mycases
```

```
##  [1]  43  56  23  67  81 150 110 122 161 238 138  74
```

```
my_list[["mypop"]]
```

```
## [1] 34226
```

2.4 Data frames

Data frames are like matrices in that they have a rectangular format consisting of columns and rows, and are like lists in that they can contain columns with multiple data types such as numeric, logical, character, and factor.

```
epi_data <- data.frame(mname, month, cases, pop, inc)
attributes(epi_data)
```

```
## $names
## [1] "mname" "month" "cases" "pop"   "inc"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
```

```
summary(epi_data)
```

```
##      mname        month           cases             pop
##  Apr    :1   Min.   : 1.00   Min.   : 23.00   Min.   :34226
##  Aug    :1   1st Qu.: 3.75   1st Qu.: 64.25   1st Qu.:34226
##  Dec    :1   Median : 6.50   Median : 95.50   Median :34226
##  Feb    :1   Mean   : 6.50   Mean   :105.25   Mean   :34226
##  Jan    :1   3rd Qu.: 9.25   3rd Qu.:141.00   3rd Qu.:34226
##  Jul    :1   Max.   :12.00   Max.   :238.00   Max.   :34226
##  (Other):6
##       inc
##  Min.   : 6.72
##  1st Qu.:18.77
##  Median :27.90
##  Mean   :30.75
##  3rd Qu.:41.20
##  Max.   :69.54
##
```

The coumns of a data frame can be accessed as list elements.

```
epi_data$mname
```

```
##  [1] Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
## Levels: Apr Aug Dec Feb Jan Jul Jun Mar May Nov Oct Sep
```

```
epi_data$inc
```

```
##  [1] 12.563548 16.361830  6.720037 19.575761 23.666219 43.826331 32.139309
##  [8] 35.645416 47.040262 69.537778 40.320224 21.620990
```

Data frames can also be accessed via matrix style subscriptiving of rows and columns.

```
epi_data[3, 4]
```

```
## [1] 34226
```

```
epi_data[1:3,]
```

```
##     mname month cases   pop       inc
## Jan   Jan     1    43 34226 12.563548
## Feb   Feb     2    56 34226 16.361830
## Mar   Mar     3    23 34226  6.720037
```

```
epi_data[,3]
```

```
##  [1]  43  56  23  67  81 150 110 122 161 238 138  74
```

```
epi_data[,"inc"]
```

```
##  [1] 12.563548 16.361830  6.720037 19.575761 23.666219 43.826331 32.139309
##  [8] 35.645416 47.040262 69.537778 40.320224 21.620990
```

Conditional statements can also be used to extract data records meeting certain conditions. Alternately, the `subset()` function can be used to select rows from the data frame using a conditional statement.

```
epi_data[epi_data$inc > 10,]
```

```
##     mname month cases   pop      inc
## Jan   Jan     1    43 34226 12.56355
## Feb   Feb     2    56 34226 16.36183
## Apr   Apr     4    67 34226 19.57576
## May   May     5    81 34226 23.66622
## Jun   Jun     6   150 34226 43.82633
## Jul   Jul     7   110 34226 32.13931
## Aug   Aug     8   122 34226 35.64542
## Sep   Sep     9   161 34226 47.04026
## Oct   Oct    10   238 34226 69.53778
## Nov   Nov    11   138 34226 40.32022
## Dec   Dec    12    74 34226 21.62099
```

```
epi_data[epi_data$mname == "Jun",]
```

```
##     mname month cases   pop      inc
## Jun   Jun     6   150 34226 43.82633
```

```
new_data <- epi_data[epi_data$month >= 7,]
new_data
```

```
##     mname month cases   pop      inc
## Jul   Jul     7   110 34226 32.13931
## Aug   Aug     8   122 34226 35.64542
## Sep   Sep     9   161 34226 47.04026
## Oct   Oct    10   238 34226 69.53778
## Nov   Nov    11   138 34226 40.32022
## Dec   Dec    12    74 34226 21.62099
```

```
new_data2 <- subset(epi_data, month >= 7)
new_data2
```

```
##     mname month cases   pop      inc
```

```
## Jul   Jul     7   110 34226 32.13931
## Aug   Aug     8   122 34226 35.64542
## Sep   Sep     9   161 34226 47.04026
## Oct   Oct    10   238 34226 69.53778
## Nov   Nov    11   138 34226 40.32022
## Dec   Dec    12    74 34226 21.62099
```

3 Tibbles

Throughout the rest of this workshop we will be working with "tibbles" instead of R's traditional "data frame". Tibbles actually *are* data frames, but they change the way data frames work to reduce some of the problems caused by the limitations of the older `data.frame` class.

If you're wondering where the name came from, tibbles originally had the class `tbl_df`, which stood for "table" and "data frame". People soon began pronouncing this new class "tibble diff", and ultimately just "tibble".

Tibbles are defined in the tibble package, which can be loaded with the library function.

```
library(tibble)
```

If you run this code and get the error message "there is no package called 'tibble' ", you'll need to first install it, then run `library()` once again.

```
install.packages("tibble")
library(tibble)
```

You only need to install a package once, but you need to reload it every time you start a new session.

3.1 Creating tibbles

Most of the functions in the tidyverse create tibbles, and all of them can accept tibbles as arguments. However many packages in R use traditional data frames. in which case you might want to coerce a data frame to a tibble using `as_tibble()`. Here is an example using the built in `iris` dataset, a data frame.

```
as_tibble(iris)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

Here you can see the built-in iris dataset has 150 rows and 5 columns.

You can create a new tibble from individual vectors with `tibble()`. `tibble()` will automatically recycle inputs of length 1, and allows you to refer to variables that you just created, as shown below.

```
df <- tibble(
  woreda = "Mecha",
  iso_week = 1:5,
  iso_year = 2017,
  cases = c(10, 12, 7, 9, 5),
  population = 5129,
  incidence = cases / population * 1000
)
df
```

```
## # A tibble: 5 x 6
##   woreda iso_week iso_year cases population incidence
##   <chr>     <int>    <dbl> <dbl>      <dbl>     <dbl>
## 1 Mecha         1     2017    10       5129     1.95
## 2 Mecha         2     2017    12       5129     2.34
## 3 Mecha         3     2017     7       5129     1.36
## 4 Mecha         4     2017     9       5129     1.75
## 5 Mecha         5     2017     5       5129     0.975
```

3.2 Printing tibbles

Like other objects, a tibble can be printed in two ways.

```
print(df)
```

```
## # A tibble: 5 x 6
##   woreda iso_week iso_year cases population incidence
##   <chr>     <int>    <dbl> <dbl>      <dbl>     <dbl>
## 1 Mecha         1     2017    10       5129     1.95
## 2 Mecha         2     2017    12       5129     2.34
## 3 Mecha         3     2017     7       5129     1.36
## 4 Mecha         4     2017     9       5129     1.75
## 5 Mecha         5     2017     5       5129     0.975
```

or simply

```
df
```

```
## # A tibble: 5 x 6
##   woreda iso_week iso_year cases population incidence
##   <chr>     <int>    <dbl> <dbl>      <dbl>     <dbl>
## 1 Mecha         1     2017    10       5129     1.95
## 2 Mecha         2     2017    12       5129     2.34
## 3 Mecha         3     2017     7       5129     1.36
## 4 Mecha         4     2017     9       5129     1.75
## 5 Mecha         5     2017     5       5129     0.975
```

There are three main differences between how a tibble is printed in your console and how a data frame is printed. We use a large dataset containing epidemiological data to illustrate some differences. First we load the data.

```
library(readr)
data <- read_csv(file = "data_day1.csv")
```

```
## Parsed with column specification:
## cols(
##   WID = col_double(),
```

```
##   woreda_name = col_character(),
##   obs_date = col_date(format = ""),
##   test_pf_tot = col_double(),
##   test_pv_only = col_double(),
##   pop_at_risk = col_double(),
##   mal_case = col_double(),
##   iso_year = col_double(),
##   iso_week = col_double(),
##   data_source = col_character()
## )
```

We won't print it here because it is too long, but to see what the data looks like in your console when printed as a `data.frame`, run this code:

```r
as.data.frame(data)
```

Unless you have a very wide screen, you probably only see the last few columns and some of the rows, along with the message `reached getOption("max.print")`. The column names and first few rows are not visible unless you scroll up in your console window quite a bit. Compare that to how `data_day1` looks when printed as a tibble:

```r
data
```

```
## # A tibble: 676 x 10
##      WID woreda_name obs_date   test_pf_tot test_pv_only pop_at_risk
##    <dbl> <chr>       <date>           <dbl>        <dbl>       <dbl>
##  1    23 Fogera      2012-07-15         696          118     209534.
##  2    23 Fogera      2012-07-22         656          122     209534.
##  3    23 Fogera      2012-07-29         551          107     209534.
##  4    23 Fogera      2012-08-05         493          106     209534.
##  5    23 Fogera      2012-08-12         430          106     209534.
##  6    23 Fogera      2012-08-19         405          107     209534.
##  7    23 Fogera      2012-08-26         384          111     209534.
##  8    23 Fogera      2012-09-02         393          117     209534.
##  9    23 Fogera      2012-09-09         411          123     209534.
## 10    23 Fogera      2012-09-16         469          135     209534.
## # ... with 666 more rows, and 4 more variables: mal_case <dbl>,
## #   iso_year <dbl>, iso_week <dbl>, data_source <chr>
```

First, only the first 10 rows of the data frame are printed. Second, only columns that fit on the screen are printed. The remainder are listed at the bottom. Third, the class of each column is given beneath its name.

As a whole, these improvements make printing tibbles much easier on the eyes than printing data frames.

3.3 Using tibbles with older functions

Some older functions don't work with tibbles. If you find yourself unable to run an older function with a tibble, use `as.data.frame()` to turn it into a traditional data frame.

```r
as.data.frame(data)
```

4 Additional topics

Before moving on, it is necessary to mention a few more features of R that will come up in our demonstrations and exercises throughout the week.

4.1 Coercing data to differnet classes

As we will see in the next section, we accomplish various tasks in R by calling functions and supplying various objects as arguments to these functions. In some cases, we may need to quickly convert data to a different object type before providing it as a function argument. We can do this by coercing the data into a new class using functions like `as.factor()`, `as.character()`, and `as.numeric()`.

```
myvector2 <- c(2012, 2012, 2012, 2013, 2013, 2013)
summary(myvector2)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2012    2012    2012    2012    2013    2013
```

```
levels(myvector2)
```

```
## NULL
```

```
summary(as.factor(myvector2))
```

```
## 2012 2013
##    3    3
```

```
levels(as.factor(myvector2))
```

```
## [1] "2012" "2013"
```

```
as.character(myvector2)
```

```
## [1] "2012" "2012" "2012" "2013" "2013" "2013"
```

4.2 Dates in R

In disease surveillance and environmental monitoring, we often need to keep track of the time when a particular data record was collected. In R, there is a special object class for storing date information.

```
today <- Sys.Date()
today
```

```
## [1] "2019-04-18"
```

```
class(today)
```

```
## [1] "Date"
```

There are variety of ways to create date objects. One simple way is to use the `as.Date()` function to coerce a text object into a date object. Date objects can also coerced back to text or numeric objects. Later in the workshop, we will use some more advance functions that can convert epidemiological weeks into dates and calculate the day of the year based on a date object.

```
datetxt <- c("2017-12-30", "2018-1-2", "2018-1-3", "2018-1-4")
summary(datetxt)
```

```
##    Length     Class      Mode
##         4 character character
```

```
class(datetxt)
```

```
## [1] "character"
```

```
dateobj <- as.Date(datetxt)
summary(dateobj)
```

```
##         Min.      1st Qu.       Median         Mean      3rd Qu. 
## "2017-12-30" "2018-01-01" "2018-01-02" "2018-01-02" "2018-01-03" 
##         Max. 
## "2018-01-04"
```

```r
class(dateobj)
```

```
## [1] "Date"
```

```r
as.character(dateobj)
```

```
## [1] "2017-12-30" "2018-01-02" "2018-01-03" "2018-01-04"
```

```r
as.numeric(dateobj)
```

```
## [1] 17530 17533 17534 17535
```

For working with epidemiological data, we might also want to find out what week or year a date falls in, following the ISO conventions. The package `lubridate` helps us with these conversion from date objects to iso weeks and iso years.

```r
library(lubridate)
```

```
## 
## Attaching package: 'lubridate'
```

```
## The following object is masked from 'package:base':
## 
##     date
```

```r
isoweek(dateobj)
```

```
## [1] 52  1  1  1
```

```r
isoyear(dateobj)
```

```
## [1] 2017 2018 2018 2018
```

4.3 Missing data - the `NA` symbol

The `NA` symbol is a special value used in R to indicate missing data. When processing and managing data, it is critical that missing data be appropriately flagged as `NA` rather treated as zero or some other arbitrary value. Most R functions have build-in methods for handing missing data, and as we will see in later examples it is often necessary to choose the most appropriate method for a particular analysis. Below are some quick examples.

```r
myvector <- c(2, NA, 9, 2, 1, NA)
is.na(myvector)
```

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

```r
sum(is.na(myvector))
```

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

```r
mean(myvector)
```

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

```r
mean(myvector, na.rm=T)
```

```
## [1] 3.5
```

5 Basic statistical analysis

Functions are also used to carry various types of statistical tests in R. For example, to compute confidence intervals for a particular variable in our data frame, we can call the `t.test()` function. Argument `x` is the data. The default confidence level is 0.95, but we can specify a value for the `conf.level` argument if we want something different.

```
demo_data <- read_csv(file = "data_day1.csv")
```

```
## Parsed with column specification:
## cols(
##   WID = col_double(),
##   woreda_name = col_character(),
##   obs_date = col_date(format = ""),
##   test_pf_tot = col_double(),
##   test_pv_only = col_double(),
##   pop_at_risk = col_double(),
##   mal_case = col_double(),
##   iso_year = col_double(),
##   iso_week = col_double(),
##   data_source = col_character()
## )
```

```
t.test(x=demo_data$mal_case)
```

```
##
##  One Sample t-test
##
## data:  demo_data$mal_case
## t = 28.676, df = 675, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  488.9965 560.8822
## sample estimates:
## mean of x
##  524.9393
```

```
t.test(x=demo_data$mal_case, conf.level=0.9)
```

```
##
##  One Sample t-test
##
## data:  demo_data$mal_case
## t = 28.676, df = 675, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 90 percent confidence interval:
##  494.7878 555.0909
## sample estimates:
## mean of x
##  524.9393
```

```
t.test(x=demo_data$mal_case, conf.level=0.99)
```

```
##
##  One Sample t-test
##
## data:  demo_data$mal_case
```

```
## t = 28.676, df = 675, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 99 percent confidence interval:
##  477.6534 572.2253
## sample estimates:
## mean of x
##  524.9393
```

How does the `t.test()` function know what confidence level to use if we do not specify the `conf.level` argument? In many cases, arguments have a default value. If the argument is no specified in the function call, then the default value is used. When using statistical fundtions in R, it is particularly important to study the documentation to learn the default values and decide if they are sufficient for the analysis.

```
help(t.test)
```

To conduct a two-sample t-test, we can call the same function and specify a formula where the variable to be compared is specified to the left of the tilde (~) symbol and a variable indicating the group to which each observation belongs is specified on the right.

```
t.test(mal_case ~ woreda_name, data=demo_data)
```

```
##
##  Welch Two Sample t-test
##
## data:  mal_case by woreda_name
## t = -11.211, df = 637.74, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -443.1602 -311.0587
## sample estimates:
## mean in group Fogera mean in group Metema
##             336.3846             713.4941
```

The following code carries out a two-way analysis of variance of outpatient malaria cases by woreda and year. The `aov()` function take two arguments, a model formula and the data object and generates an object of class aov that contains the results of the analysis. Note that we use the `as.factor()` function to coerce epi_year into a factor. We the supply the aov object as an argument to the `anova()` function to produce a standard analysis of variance table. We can also supply it as an argument to the `TukeyHSD()` function to generate a multiple comparisons object. This object can then be supplied as an argument to the `print()` and `plot()` functions to view the results.

```
demo_aov <- aov(mal_case ~ woreda_name + as.factor(iso_year), data=demo_data)
class(demo_aov)
```

```
## [1] "aov" "lm"
```

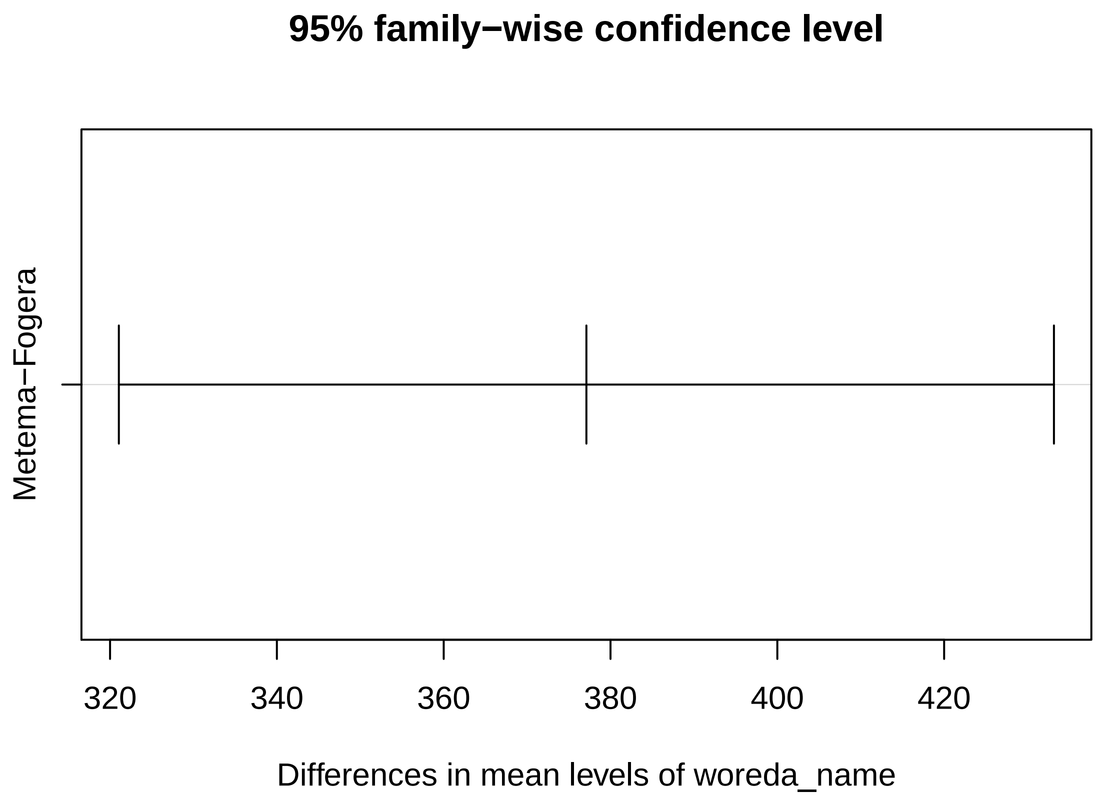
```
anova(demo_aov)
```

```
## Analysis of Variance Table
##
## Response: mal_case
##                      Df   Sum Sq  Mean Sq F value    Pr(>F)
## woreda_name           1 24033752 24033752 174.482 < 2.2e-16 ***
## as.factor(iso_year)   6 36858745  6143124  44.598 < 2.2e-16 ***
## Residuals           668 92012397   137743
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

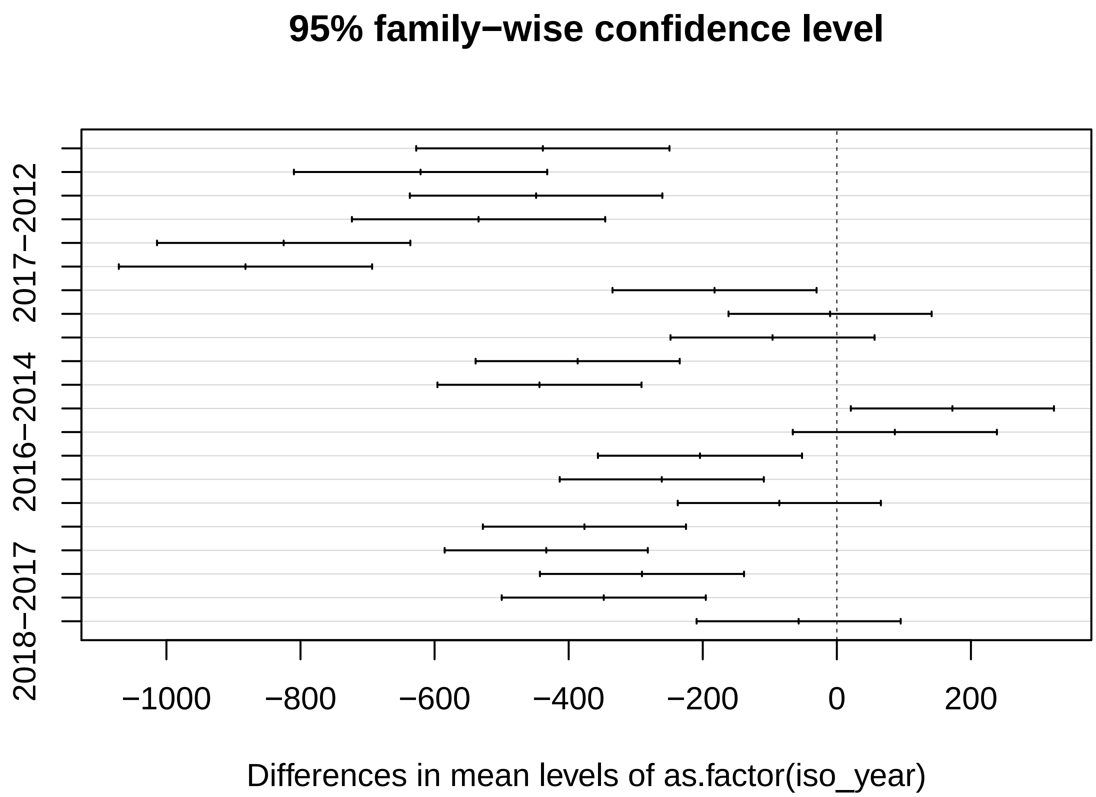
```r
demo_mc <- TukeyHSD(demo_aov)
print(demo_mc)
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = mal_case ~ woreda_name + as.factor(iso_year), data = demo_data)
##
## $woreda_name
##                    diff      lwr      upr p adj
## Metema-Fogera 377.1095 321.0528 433.1661     0
##
## $`as.factor(iso_year)`
##                  diff         lwr        upr     p adj
## 2013-2012 -438.51500  -627.39952 -249.63048 0.0000000
## 2014-2012 -620.93808  -809.82259 -432.05356 0.0000000
## 2015-2012 -448.60226  -636.90734 -260.29719 0.0000000
## 2016-2012 -534.42846  -723.31298 -345.54395 0.0000000
## 2017-2012 -825.11115 -1013.99567 -636.22664 0.0000000
## 2018-2012 -882.08231 -1070.96682 -693.19779 0.0000000
## 2014-2013 -182.42308  -334.63056  -30.21560 0.0076487
## 2015-2013  -10.08726  -161.57508  141.40056 0.9999951
## 2016-2013  -95.91346  -248.12094   56.29402 0.5054255
## 2017-2013 -386.59615  -538.80363 -234.38867 0.0000000
## 2018-2013 -443.56731  -595.77479 -291.35983 0.0000000
## 2015-2014  172.33581    20.84799  323.82363 0.0141951
## 2016-2014   86.50962   -65.69787  238.71710 0.6291514
## 2017-2014 -204.17308  -356.38056  -51.96560 0.0015607
## 2018-2014 -261.14423  -413.35171 -108.93675 0.0000105
## 2016-2015  -85.82620  -237.31402   65.66162 0.6327339
## 2017-2015 -376.50889  -527.99671 -225.02107 0.0000000
## 2018-2015 -433.48004  -584.96786 -281.99222 0.0000000
## 2017-2016 -290.68269  -442.89017 -138.47521 0.0000005
## 2018-2016 -347.65385  -499.86133 -195.44637 0.0000000
## 2018-2017  -56.97115  -209.17863   95.23633 0.9260030
```

```r
plot(demo_mc)
```



95% family−wise confidence level
Metema−Fogera
320
340
360
380
400
420
Differences in mean levels of woreda_name



The following code carries out a linear regression analysis of positive tests for *Plasmodium vivax* only (dependent variable) as a function of positive tests for *Plasmodium falciparum*/mixed (independent variable). As with the `aov()` function, the `lm()` function takes a formula and a data argument and returns an object belonging to class `lm`. We can then use various other functions to summarize the lm object in various ways and to extract coefficients, fitted values, and residuals.

```
demo_lm <- lm(sqrt(test_pv_only + 1) ~ sqrt(test_pf_tot + 1), data=demo_data)
class(demo_lm)
```

```
## [1] "lm"
```

```
attributes(demo_lm)
```

```
## $names
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
##
## $class
## [1] "lm"
```

```
summary(demo_lm)
```

```
##
## Call:
## lm(formula = sqrt(test_pv_only + 1) ~ sqrt(test_pf_tot + 1),
##     data = demo_data)
##
```

```
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -6.2019 -2.1385 -0.2296  1.7210 13.5760 
## 
## Coefficients:
##                      Estimate Std. Error t value Pr(>|t|)    
## (Intercept)           3.15264    0.23155   13.62   <2e-16 ***
## sqrt(test_pf_tot + 1)  0.35718    0.01136   31.44   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 2.863 on 674 degrees of freedom
## Multiple R-squared:  0.5946, Adjusted R-squared:  0.594 
## F-statistic: 988.5 on 1 and 674 DF,  p-value: < 2.2e-16
```

```
anova(demo_lm)
```

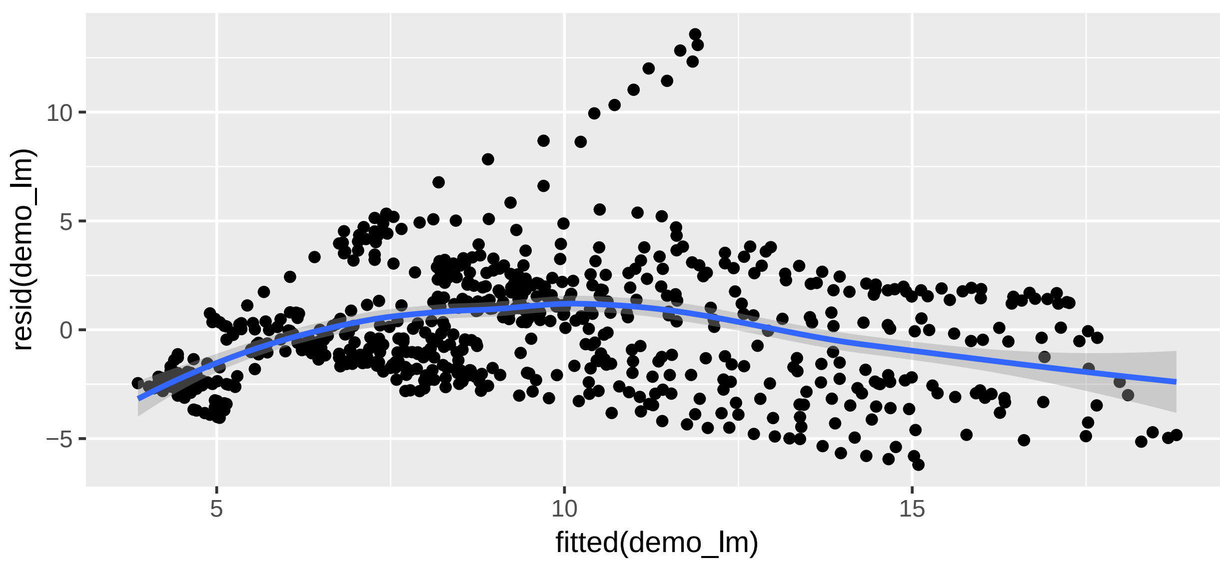
```
## Analysis of Variance Table
## 
## Response: sqrt(test_pv_only + 1)
##                        Df Sum Sq Mean Sq F value    Pr(>F)    
## sqrt(test_pf_tot + 1)   1 8104.7  8104.7  988.53 < 2.2e-16 ***
## Residuals             674 5526.0     8.2                      
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
coef(demo_lm)
```

```
##           (Intercept) sqrt(test_pf_tot + 1) 
##              3.152635              0.357179
```

```
library(ggplot2)
qplot(x=fitted(demo_lm), y=resid(demo_lm), geom=c("point", "smooth"))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



6 Day 1 exercises

Run through the demonstration yourself by typing all of the commands above into a script on your computer, running each line of the script, and then examining the output on your computer. Whenever a new function is introduced, use the `help()` function to look it up and examine the various arguments that can be specified. Experiment with changing some of the arguments to see how they affect the output of the function.

Then, to pratice what you have learned, try out the following exercises:

The following set of 24 numbers represents *Plasmodium falciparum* cases collected from January 2015 through December 2016 at a health center. Create a vector called `pfcases` containing these numbers.

81 55 79 107 135 210 166 175 186 355 228 126 81 65 53 62 54 88 107 93 115 157 94 54

The following set of 24 numbers represents *Plasmodium vivax* cases collected from January 2015 through December 2016 at the same health center. Create a vector called `pvcases` containing these numbers.

51 43 66 75 89 146 120 99 115 186 150 86 61 50 33 22 45 76 79 70 80 105 74 33

Using these data, write R code to carry out the following tasks:

1. Calculate the sum of all *Plasmodium vivax* cases over the two years.
2. Calculate the mean number of monthly *Plasmodium falciparum* over the two years.
3. Generate a logical vector that indicates which elements of `pfcases` have a value greater than or equal to 100.
4. Generate a vector that contains the number of the month (integer values 1 through 12) corresponding to each element of `pfcases` and `pvcases`.
5. Generate a vector that contains the number of the year (2015 or 2016) corresponding to each element of pfcases and pvcases.
6. Generate a new vector that contains the population corresponding to each element of `pfcases` and `pvcases`, assuming a population of 23,000 in 2015 and 23,500 in 2016.
7. Generate a new vector that contains the total number of malaria cases in each month.
8. Generate a new vector that contains the total malaria incidence (per 1,000 population) in each month.
9. Combine all of the vectors that you have created to make a data frame.
10. Use the `subset()` function to create a new data frame that contains only data for 2016.
11. Use the lm function to carry out a linear regression with pfcases as the dependent variable and pvcases as the independent variable.