Getting Started in Base R

Saghir Bashir, Dirk Eddelbuettel

This version was compiled on October 2, 2018

Are you curious to learn what R can do for you? Do you want to see how it works? Yes, then this "Getting Started" guide is for you. It uses realistic examples and a real life dataset to manipulate, visualise and summarise data. By the end of it you will have an overview of the key concepts of R.

R | Statistics | Data Science

1. Preface

This "Getting Started" guide will give you a flavour of what R[1] can do for you. To get the most out of this guide, read it whilst doing the examples and exercises using RStudio[2]^.

Experiment Safely. Be brave and experiment with commands and options as it is an essential part of the learning process. Things can (and will) go "wrong", like, getting error messages or deleting things that you create by using this guide. You can recover from most situations (e.g. by restarting R). To do this "safely" start with a *fresh* R session without any other data loaded (otherwise you could lose it).

2. Introduction

Before Starting. Make sure that:

1. R and RStudio are installed.
2. http://ilustat.com/shared/Getting-Started-in-R.zip has been downloaded and unzipped
3. Double click `"Getting-Started-in-R.Rproj"` to open RStudio with the setup for this guide.

Starting R & RStudio. R starts automatically when you open RStudio (see Figure 1). The console starts with information about the version number, license and contributors. The last line is a standard prompt ">" that indicates R is ready and expecting instructions to do something.

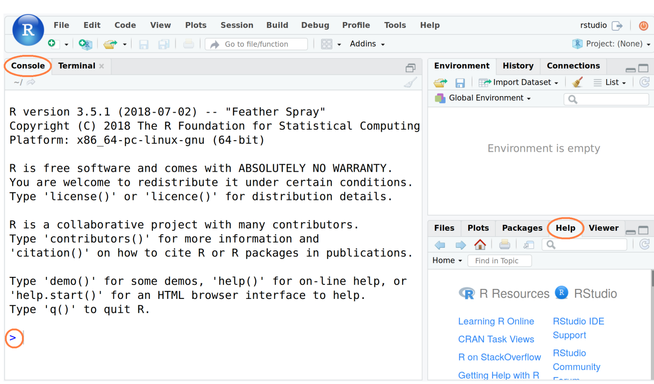


Fig. 1. RStudio Screenshot with Console on the left and Help tab in the bottom right

Quitting R & RStudio. When you quit RStudio you will be asked whether to `Save workspace` with two options:

- "Yes" – Your current R workspace (containing the work that you have done) will be restored next time you open RStudio.
- "No" – You will start with a fresh R session next time you open RStudio. For now select "*No*" to prevent errors being carried over from previous sessions).

3. R Help

I strongly recommend that you learn how to use R's useful and extensive built-in help system which is an essential part of finding solutions to your R programming problems.

`help()` function. From the R "Console" you can use the `help()` function or `?`. For example, try the following two commands (which give the same result):

```
help(mean)
?mean
```

Keyword search. To do a keyword search use the function `apropos()` with the keyword in double quotes (`"keyword"`) or single quote (`'keyword'`). For example:

```
apropos("mean")
#   [1] ".colMeans"       ".rowMeans"
#   [3] "colMeans"        "kmeans"
#   [5] "mean"            "mean_cl_boot"
#   [7] "mean_cl_normal"  "mean_sdl"
#   [9] "mean_se"         "mean.Date"
#  [11] "mean.default"    "mean.difftime"
#  [13] "mean.POSIXct"    "mean.POSIXlt"
#  [15] "rowMeans"        "weighted.mean"
```

Help Examples. Use the `example()` function to run the examples at the end of the help for a function:

```
example(mean)
#
#  meanR> x <- c(0:10, 50)
#
#  meanR> xm <- mean(x)
#
#  meanR> c(xm, mean(x, trim = 0.10))
#  [1] 8.75 5.50
```

RStudio Help. Rstudio provides search box in the "Help" tab to make your life easier (see Figure 1).

Searching On-line For R Help. There are a lot of on-line resources that can help. However you must understand that blindly copying and pasting could be harmful and further it won't help you to learn and develop. When you search on-line use `[R]` in your search term (e.g. "`[R] summary statistics by group`"). Note that often there is more than one solution to your problem. It is good to investigate the different options.

[1] R project: https://www.r-project.org/
[2] RStudio IDE: https://www.rstudio.com/products/RStudio/

Exercise. Try the following:

1. `help(median)`,
2. `?sd`
3. `?max`

4. Some R Concepts

In R speak, scalars, vectors/variables and datasets are called ***objects***. To create objects (things) we have to use the assignment operator "<-". For example, below, object `height` is assigned a value of 173 (typing `height` shows its value):

```
height <- 173
height
#  [1] 173
```

Warning: R is case sensitive. `age` and `AgE` are different:

```
age <- 10
AgE <- 50
```

```
age
#  [1] 10
AgE
#  [1] 50
```

New lines. R commands are usually separated by a new line but they can also be separated by a semicolon ";".

```
Name <- "Leo"; Age <- 25; City <- "Lisbon"
Name; Age; City
#  [1] "Leo"
#  [1] 25
#  [1] "Lisbon"
```

Comments. It is useful to put human readable comments in your programs. These comments could help the future you when you go back to your program. R comments start with a hash sign (`#`). Everything after the hash to the end of the line will be ignored by R.

```
# This comment line will be ignored when run.
City      # Text after "#" is ignored.
#  [1] "Lisbon"
```

Warning. If an R command is not complete then R will show a plus sign ("+") prompt on second and subsequent lines until the command syntax is correct.

```
+
```

To break out this, press the escape key (`ESC`).

Recalling previous commands. To recall a previously typed commands use the up arrow key (↑). To go between previously typed commands use the up and down arrow (↓) keys. Once a command is recalled, it can be modified/corrected using the left (←) and right arrow (→) keys.

5. R as a Calculator

You can use R as a calculator. Try the following:

```
2 + 3
#  [1] 5
(5*11)/4 - 7
#  [1] 6.75
# ^ = "to the power of"
7^3
#  [1] 343
```

Other math functions. You can also use standard mathematical functions that are typically found on a scientific calculator.

- Trigonometric: `sin()`, `cos()`, `tan()`, `acos()`, `asin()`, `atan()`
- Rounding: `abs()`, `ceiling()`, `floor()`, `round()`, `sign()`, `signif()`, `sqrt()`, `trunc()`
- Logarithms & Exponentials: `exp()`, `log()`, `log10()`, `log2()`

```
# Square root
sqrt(2)
#  [1] 1.41421
# Round down to nearest integer
floor(8.6178)
#  [1] 8
# Round to 2 decimal places
round(8.6178, 2)
#  [1] 8.62
```

Exercise. What do the following pairs of examples do?

1. `ceiling(18.33)` and `signif(9488, 2)`
2. `exp(1)` and `log10(1000)`
3. `sign(-2.9)` and `sign(32)`
4. `abs(-27.9)` and `abs(11.9)`

6. Some More R Concepts

You can do some clever and useful things with using the assignment operator "<-":

```
roomLength <- 7.8
roomWidth <- 6.4
roomArea <- roomLength * roomWidth
roomArea
#  [1] 49.92
```

Text objects. You can also assign text to an objects.

```
Greeting <- "Hello World!"
Greeting
#  [1] "Hello World!"
```

Vectors. The objects presented so far have all been scalars (single values). Working with vectors is where R shines best as they are the basic building blocks of datasets. To create a vector we can use the `c()` (combine values into a vector) function.

```
# A "numeric" vector
x1 <- c(26, 10, 4, 7, 41, 19)
x1
#  [1] 26 10  4  7 41 19
# A "character" vector of country names
x2 <- c("Peru", "Italy", "Cuba", "Ghana")
```

```
x2
#  [1] "Peru"  "Italy" "Cuba"  "Ghana"
```

There are many other ways to create vectors, for example, `rep()` (replicate elements) and `seq()` (create sequences):

```
# Repeat vector (2, 6, 7, 4) three times
r1 <- rep(c(2, 6, 7, 4), times=3)
r1
#   [1] 2 6 7 4 2 6 7 4 2 6 7 4
# Vector from -2 to 3 incremented by half
s1 <- seq(from=-2, to=3, by=0.5)
s1
#   [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5
#   [9]  2.0  2.5  3.0
```

Vector operations. You can do also calculations on vectors, for example using `x1` from above:

```
x1 * 2
#  [1] 52 20  8 14 82 38
round(sqrt(x1*2.6), 2)
#  [1]  8.22  5.10  3.22  4.27 10.32  7.03
```

Missing Values. Missing values are coded as `NA` in R. For example,

```
x2 <- c(3, -7, NA, 5, 1, 1)
x2
#  [1]  3 -7 NA  5  1  1
x3 <- c("Rat", NA, "Mouse", "Hamster")
x3
#  [1] "Rat"     NA        "Mouse"   "Hamster"
```

Managing Objects. Use function `ls()` to list the objects in your workspace. The `rm()` function removes (delete) them.

```
ls()
#   [1] "age"        "Age"        "AgE"
#   [4] "City"       "Greeting"   "height"
#   [7] "Name"       "p"          "r1"
#  [10] "roomArea"   "roomLength" "roomWidth"
#  [13] "s1"         "x"          "x1"
#  [16] "x2"         "x3"         "xm"
rm(x, x1, x2, x3, xm, r1, s1, AgE, age)
ls()
#  [1] "Age"        "City"       "Greeting"
#  [4] "height"     "Name"       "p"
#  [7] "roomArea"   "roomLength" "roomWidth"
```

Exercise. Calculate the `gross` by adding the `tax` to `net` amount.

```
net <- c(108.99, 291.42, 16.28, 62.29, 31.77)
tax <- c(22.89, 17.49, 0.98, 13.08, 6.67)
```

7. R Functions and Packages

R Functions. We have already used some R functions (e.g. `c()`, `mean()`, `rep()`, `sqrt()`, `round()`). Most of the computations in R involves using functions. A function essentially has a name and a list of arguments separated by a comma. Let's have look at an example:

```
seq(from = 5, to = 8, by = 0.4)
#  [1] 5.0 5.4 5.8 6.2 6.6 7.0 7.4 7.8
```

The function name is `seq` and it has three arguments `from`, `to` and `by`. The arguments `from` and `to` are the start and end values of a sequence that you want to create, and `by` is the increment of the sequence. The `seq()` functions has other arguments that you could use which are documented in the help page. For example, we could use the argument `length.out` (instead of `by`) to fix the length of the sequence as follows:

```
seq(from = 5, to = 8, length.out = 16)
#   [1] 5.0 5.2 5.4 5.6 5.8 6.0 6.2 6.4 6.6 6.8
#  [11] 7.0 7.2 7.4 7.6 7.8 8.0
```

Custom Functions. You can create your own functions (using the `function()` function) which is a very powerful way to extend R. Writing your own functions is outside the scope of this guide. As you get more and more familiar with R it is very likely that you will need to learn do so but for now you don't need to.

R Packages. You can do many things with a standard R installation and it can be extended using contributed packages. Packages are like apps for R. They can contain functions, data and documentation.

Base R. Base R already comes with over two-thousand functions that have been proven to be versatile, reliable and stable. That is no small feat. When it is possible to solve a problem with *fewer* external dependencies, doing so follows time-honoured best practices. Think carefully before adding dependencies,

`tinyverse`. The philosophy of *less is more* is at the core of the tinyverse[3]. Fewer dependencies means smaller footprint, faster installation, and most importantly fewer nodes in your dependency graph. Experience, as well as both empirical and theorectical software engineering practice have demonstrated that failure increases with complexity.

So choosing when to rely on additional packages has to balance the increased functionality a package brings with both its history of development, its development model, maintenance status and history of both changes and fixes. This *is* complex, and there are no easy answers. By adding another package, we open a door to interface changes we no longer control. It is clearly valuable at times, yet one has to remain aware of the costs that may accrue as a consequence. So this document takes the few that *fewer is better*, and will rely on only two additional packages: `data.table` for data wrangling as well as input/output, and `ggplot2` for visualization.

Installation. If needed, install these two packages via

```
install.packages(c("data.table", "ggplot2"))
```

8. Chick Weight Data

R comes with many datasets installed[4]. We will use the `ChickWeight` dataset to learn about the tidyverse. The help system gives a basic summary of the experiment from which the data was collect:

[3] Tinyverse: http://www.tinyverse.org/

[4] Type `data()` in the R console to see a list of the datasets.

"The body weights of the chicks were measured at birth and every second day thereafter until day 20. They were also measured on day 21. There were four groups of chicks on different protein diets."

You can get more information, including references by typing:

```
help("ChickWeight")
```

The Data. There are 578 observations (rows) and 4 variables:

- `Chick` – unique ID for each chick.
- `Diet` – one of four protein diets.
- `Time` – number of days since birth.
- `weight` – body weight of chick in grams.

Note. `weight` has a lower case `w` (recall R is case sensitive).

Objective. Investigate the *effect of diet on the weight over time*.

9. Importing The Data

First we will import the data from a file called `ChickWeight.csv` using the `read_csv()` function from the `readr` package (part of the `tidyverse`). The first thing to do, outside of R, is to open the file `ChickWeight.csv` to check what it contains and that it makes sense. Now we can import the data as follows:

```
suppressMessages(library(data.table))  # tinyverse
cw <- fread("ChickWeight.csv")
```

All columns (variables) have been read in as numeric values (i.e. `col_double()`) but you may see that they are read in a integer (i.e. `col_int()`) due to operating system differences. **FIXME**

Important Note. If all goes well then the data is now stored in an R object called `CW`. If you get the following error message then you need to change the working directory to where the data is stored.

```
Error: 'ChickWeight.csv' does not exist in current
working directory ...
```

Change the working directory in RStudio. From the menu bar select "Session - Set Working Directory - Choose Directory..." then go to the directory where the data is stored. Alternatively, within in R, you could use the function `setwd()`[5].

10. Looking at the Dataset

To look at the data type just type the object (dataset) name:

```
cw
#       Chick Diet Time weight
#    1:    18    1    0     39
#    2:    18    1    2     35
#    3:    16    1    0     41
#    4:    16    1    2     45
#    5:    16    1    4     49
#   ---
#  574:    48    4   14    170
#  575:    48    4   16    222
#  576:    48    4   18    261
#  577:    48    4   20    303
#  578:    48    4   21    322
```

[5] Use `getwd()` to see the current working directory and `setwd("/to/data/path/data.csv")` to change it (important to use / even for Microsoft Windows).

`glimpse()` function FIXME. If there are too many variables then not all them may be printed. To overcome this issue we can use the `glimpse()` function which makes it possible to see every column in your dataset (called a "data frame" in R speak).

Note that `glimpse()` gives the same result on our `data.table` object `cw`. Alternatively, the base R function `summary()` is very helpful:

```
summary(cw)
#      Chick           Diet
#   Min.   : 1.0   Min.   :1.00
#   1st Qu.:13.0   1st Qu.:1.00
#   Median :26.0   Median :2.00
#   Mean   :25.8   Mean   :2.24
#   3rd Qu.:38.0   3rd Qu.:3.00
#   Max.   :50.0   Max.   :4.00
#       Time          weight
#   Min.   : 0.0   Min.   : 35
#   1st Qu.: 4.0   1st Qu.: 63
#   Median :10.0   Median :103
#   Mean   :10.7   Mean   :122
#   3rd Qu.:16.0   3rd Qu.:164
#   Max.   :21.0   Max.   :373
```

Interpretation. Both of these show that the dataset has 578 observations and 4 variables as we would expect and as compared to the original data file `ChicWeight.csv`. So a good start.

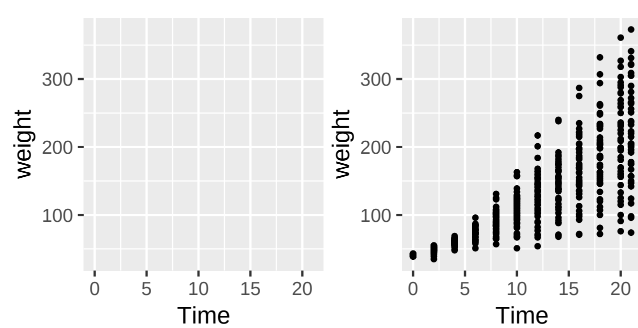
Exercise. It is important to look at the last observations of the dataset as it could reveal potential data issues. Use the `tail()` function to do this. Is it consistent with the original data file `ChickWeight.csv`?

11. Chick Weight: Data Visualisation

`ggplot2` Package. To visualise the chick weight data, we will use the `ggplot2` package (part of the `tidyverse`). Our interest is in seeing how the *weight changes over time for the chicks by diet*. For the moment don't worry too much about the details just try to build your own understanding and logic. To learn more try different things even if you get an error messages.

First plot. Let's plot the weight data (vertical axis) over time (horizontal axis).

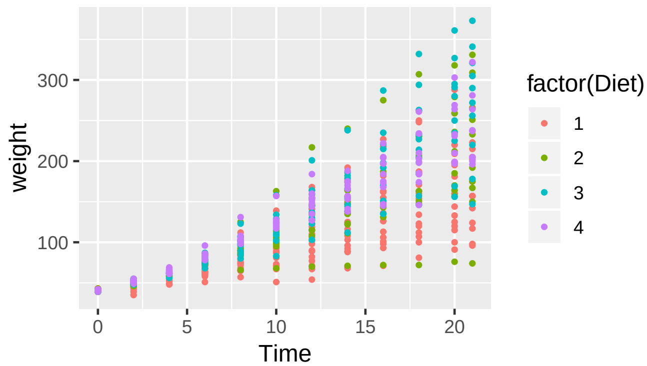
```
# An empty plot (the plot on the left)
ggplot(cw, aes(Time, weight))
# With data (the plot on the right)
ggplot(cw, aes(Time, weight)) + geom_point()
```



Exercise. Switch the variables `Time` and `weight` in code used for the plot on the right? What do you think of this new plot compared to the original?

Add colour for `Diet`. The graph above does not differentiate between the diets. Let's use a different colour for each diet.

```
# Adding colour for diet
ggplot(cw, aes(Time,weight,colour=factor(Diet))) +
  geom_point()
```



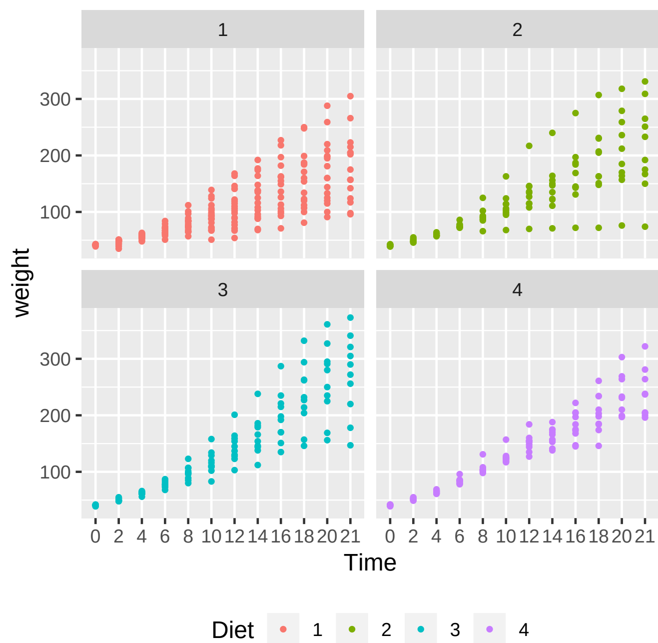
Interpretation. It is difficult to conclude anything from this graph as the points are printed on top of one another (with diet 1 underneath and diet 4 at the top).

Factor Variables. Before we continue, we have to make an important change to the `CW` dataset by making `Diet` and `Time` *factor variables*. This means that R will treat them as categorical variables instead of continuous variables. It will simplify our coding. The next section will explain the `:=` assignment.

```
cw1 <- copy(cw)
cw1[, Diet := factor(Diet)]  # update by reference
cw1[, Time := factor(Time)]
summary(cw1) # notice the difference ?
#       Chick        Diet       Time
#   Min.   : 1.0   1:220   0      : 50
#   1st Qu.:13.0   2:120   2      : 50
#   Median :26.0   3:120   4      : 49
#   Mean   :25.8   4:118   6      : 49
#   3rd Qu.:38.0           8      : 49
#   Max.   :50.0           10     : 49
#                          (Other):282
#       weight
#   Min.   : 35
#   1st Qu.: 63
#   Median :103
#   Mean   :122
#   3rd Qu.:164
#   Max.   :373
#
```

`facet_wrap()` function. To plot each diet separately in a grid using `facet_wrap()`:

```
# Adding jitter to the points
ggplot(cw1, aes(Time, weight, colour=Diet)) +
  geom_point() +
  facet_wrap(~Diet) +
  theme(legend.position = "bottom")
```



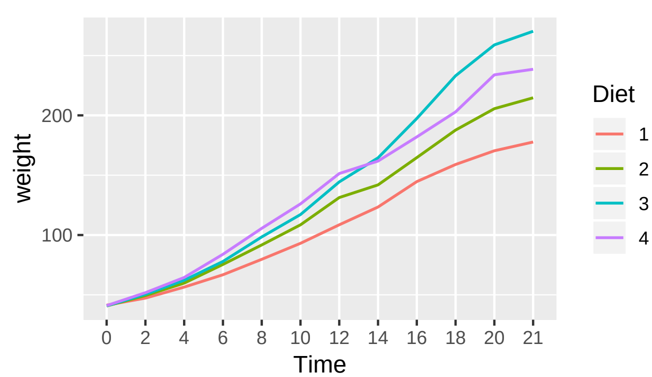
Exercise. To overcome the issue of ovelapping points we can ***jitter*** the points using `geom_jitter()`. Replace the `geom_point()` above with `geom_jitter()`. What do you observe?

Interpretation. Diet 4 has the least variability but we can't really say anything about the mean effect of each diet although diet 3 seems to have the highest.

Exercise. For the `legend.position` try using "top", "left" and "none". Do we really need a legend for this plot?

Mean line plot. Next we will plot the mean changes over time for each diet using the `stat_summary()` function:

```
ggplot(cw1, aes(Time, weight,
                group=Diet, colour=Diet)) +
  stat_summary(fun.y="mean", geom="line")
```

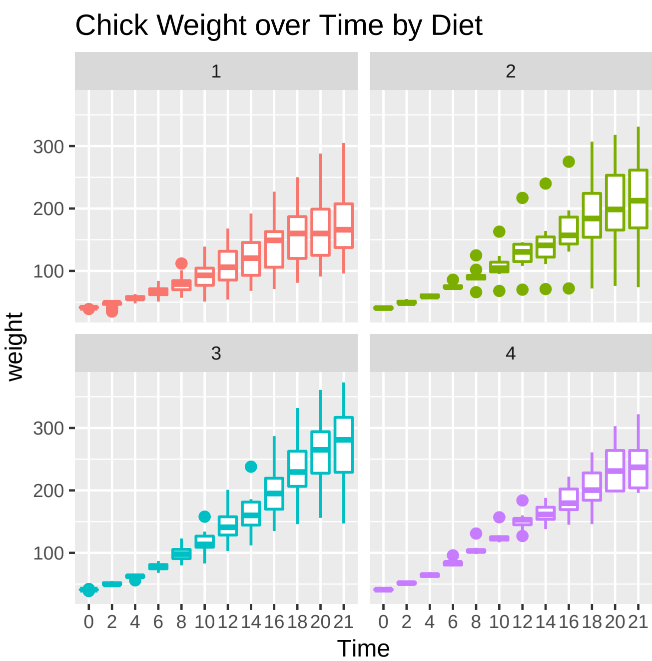


Interpretation. We can see that diet 3 has the highest mean weight gains by the end of the experiment but we don't have any information about the variation (uncertainty) in the data.

Exercise. What happens when you add `geom_point()` to the plot above? Don't forget the `+`. Does it make a difference if you put it before or after the `stat_summary(...)` line? Hint: Look very carefully at how the graph is plotted.

Box-whisker plot. To see variation between the different diets we use `geom_boxplot` to plot a box-whisker plot. A note of caution is that the number of chicks per diet is relatively low to produce this plot.

```
ggplot(cw1, aes(Time, weight, colour=Diet)) +
  facet_wrap(~ Diet) +
  geom_boxplot() +
  theme(legend.position = "none") +
  ggtitle("Chick Weight over Time by Diet")
```



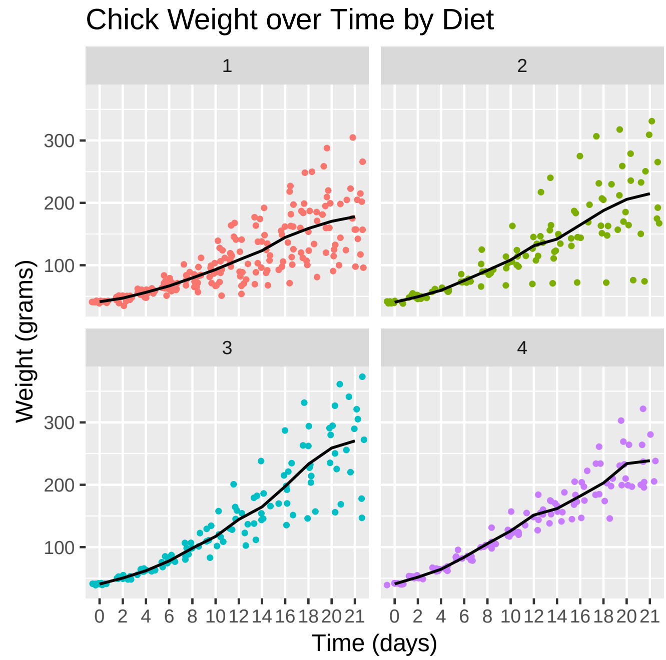
Interpretation. Diet 3 seems to have the highest "average" weight gain but it has more variation than diet 4 which is consistent with our findings so far.

Exercise. Add the following information to the the above plot:

- x-axis label (use `xlab()`): "Time (days)"
- y-axis label (use `ylab()`): "Weight (grams)"

Final Plot. Let's finish with a plot that you might include in a publication.

```
ggplot(cw1, aes(Time, weight, group=Diet,
                             colour=Diet)) +
  facet_wrap(~ Diet) +
  geom_jitter() +
  stat_summary(fun.y="mean", geom="line",
               colour="black") +
  theme(legend.position = "none") +
  ggtitle("Chick Weight over Time by Diet") +
  xlab("Time (days)") +
  ylab("Weight (grams)")
```



12. `data.table` Data Wrangling Basics

In this section we will learn how to wrangle (manipulate) datasets using the `data.table` package. **FIXME** Let's start with the `mutate()`, `select()`, `rename()`, `filter()` and `arrange()` functions.

`j` to select (or transform) columns. Adds a new variable (column) or modifies an existing one. We already used this above to create factor variables.

```
cw2 <- copy(cw)
# Added a column
cw2[, weightKg := weight/1000 ][]
#        Chick Diet Time weight weightKg
#     1:    18    1    0     39    0.039
#     2:    18    1    2     35    0.035
#    ---
#   577:    48    4   20    303    0.303
#   578:    48    4   21    322    0.322
# Modify an existing column
cw2[, Diet := paste("Diet", Diet) ][]
#        Chick   Diet Time weight weightKg
#     1:    18 Diet 1    0     39    0.039
#     2:    18 Diet 1    2     35    0.035
#    ---
#   577:    48 Diet 4   20    303    0.303
#   578:    48 Diet 4   21    322    0.322
```

`j` to select (or transform) columns. Keeps, drops or reorders variables.

```
# Drop the weight variable from CWm1 using minus
cw2[, -c("weight")]
#        Chick   Diet Time weightKg
#     1:    18 Diet 1    0    0.039
#     2:    18 Diet 1    2    0.035
#    ---
```

```
#  577:    48 Diet 4   20    0.303
#  578:    48 Diet 4   21    0.322
# Keep variables Time, Diet and weightKg
cw2[, .(Chick, Time, Diet, weightKg)]
#      Chick Time   Diet weightKg
#    1:    18    0 Diet 1    0.039
#    2:    18    2 Diet 1    0.035
#   ---
#  577:    48   20 Diet 4    0.303
#  578:    48   21 Diet 4    0.322
```

setnames() to name or rename. Renames variables whilst keeping all variables.

```
setnames(cw2, c("Diet", "weight"),
         c("Group", "Weight"))
cw2[]
#      Chick  Group Time Weight weightKg
#    1:    18 Diet 1    0     39    0.039
#    2:    18 Diet 1    2     35    0.035
#   ---
#  577:    48 Diet 4   20    303    0.303
#  578:    48 Diet 4   21    322    0.322
```

i operator. Keeps or drops observations (rows).

```
cw2[ Time == 21 & Weight > 300, ]
#    Chick  Group Time Weight weightKg
#  1:     7 Diet 1   21    305    0.305
#  2:    29 Diet 2   21    309    0.309
#  3:    21 Diet 2   21    331    0.331
#  4:    32 Diet 3   21    305    0.305
#  5:    40 Diet 3   21    321    0.321
#  6:    34 Diet 3   21    341    0.341
#  7:    35 Diet 3   21    373    0.373
#  8:    48 Diet 4   21    322    0.322
```

For comparing values in vectors use: < (less than), > (greater than), <= (less than and equal to), >= (greater than and equal to), == (equal to) and != (not equal to). These can be combined logically using & (and) and | (or).

Keying observations. Changes the order of the observations (rows).

```
cw2[, on=.(weight)] # on the fly
#      Chick  Group Time Weight weightKg
#    1:    18 Diet 1    0     39    0.039
#    2:    18 Diet 1    2     35    0.035
#   ---
#  577:    48 Diet 4   20    303    0.303
#  578:    48 Diet 4   21    322    0.322
setkey(cw2, Chick, Time) # keyed
cw2
#      Chick  Group Time Weight weightKg
#    1:     1 Diet 1    0     42    0.042
#    2:     1 Diet 1    2     51    0.051
#   ---
#  577:    50 Diet 4   20    264    0.264
#  578:    50 Diet 4   21    264    0.264
```

Exercise. What does the desc() do? Try using desc(Time). **FIXME**

13. Chaining

In reality you will end up doing multiple data wrangling steps that you want to save. Neither are optimal as they have their own issues. This is where the 'chaining' of data.table operations comes to the rescue:

```
cw3 <- copy(cw)
cw3 <- cw3[ Time %in% c(0,21)][
  , Weight := weight][
  , Group := factor(paste("Diet", Diet))][
  , .(Chick, Group, Time, Weight)][
  , on=.(Chick,Time)]
head(cw3, 10)
#      Chick  Group Time Weight
#   1:    18 Diet 1    0     39
#   2:    16 Diet 1    0     41
#   3:    15 Diet 1    0     41
#   4:    13 Diet 1    0     41
#   5:    13 Diet 1   21     96
#   6:     9 Diet 1    0     42
#   7:     9 Diet 1   21     98
#   8:    20 Diet 1    0     41
#   9:    20 Diet 1   21    117
#  10:    10 Diet 1    0     41
```

14. Chick Weight: Summary Statistics

From the data visualisations above we concluded that the diet 3 has the highest mean and diet 4 the least variation. In this section, we will quantify the effects of the diets using summmary statistics. We start by looking at the number of observations and the mean of weight grouped by **diet** and **time**.

```
cw4 <- copy(cw)
cwSum <- cw4[ , .(N = .N, Mean = mean(weight)),
             by=.(Diet, Time)]
head(cwSum)
#    Diet Time  N    Mean
#  1:    1    0 20 41.4000
#  2:    1    2 20 47.2500
#  3:    1    4 19 56.4737
#  4:    1    6 19 66.7895
#  5:    1    8 19 79.6842
#  6:    1   10 19 93.0526
```

by() function. For each distinct combination of Diet and Time, the chick weight data is summarised into the number of observations (N, using the internal variable .N denoting current group size) and the mean (Mean) of weight.

Other summaries. Let's also calculate the standard deviation, median, minimum and maximum values but only at days 0 and 21.

```
cw4 <- copy(cw)
cw4 <- cw4[ Time %in% c(0,21)]
CWS <- cw4[,.(N      = .N,
              Mean   = mean(weight),
              SDev   = sd(weight),
```

```
              Median = median(weight),
              Min    = min(weight),
              Max    = max(weight)  ),
           by=.(Diet, Time)]
options(digits=4)  ## tighter display here
CWS
#     Diet Time  N  Mean    SDev Median Min Max
#  1:    1    0 20  41.4  0.9947   41.0  39  43
#  2:    1   21 16 177.8 58.7021  166.0  96 305
#  3:    2    0 10  40.7  1.4944   40.5  39  43
#  4:    2   21 10 214.7 78.1381  212.5  74 331
#  5:    3    0 10  40.8  1.0328   41.0  39  42
#  6:    3   21 10 270.3 71.6225  281.0 147 373
#  7:    4    0 10  41.0  1.0541   41.0  39  42
#  8:    4   21  9 238.6 43.3478  237.0 196 322
```

Let's make the summaries "prettier" for a report or publication.

```
CWS[, Mean_SD := paste0(format(Mean,digits=1)," (",
                        format(SDev,digits=2),
                        ")")]
CWS[, Range := paste(Min, "-", Max)]
prettyCWSum <- CWS[, .(Diet, Time, N, Mean_SD,
                       Median, Range),
                   on=.(Diet, Time)]
prettyCWSum
#     Diet Time  N     Mean_SD Median     Range
#  1:    1    0 20  41 ( 0.99)   41.0   39 - 43
#  2:    1   21 16 178 (58.70)  166.0  96 - 305
#  3:    2    0 10  41 ( 1.49)   40.5   39 - 43
#  4:    2   21 10 215 (78.14)  212.5  74 - 331
#  5:    3    0 10  41 ( 1.03)   41.0   39 - 42
#  6:    3   21 10 270 (71.62)  281.0 147 - 373
#  7:    4    0 10  41 ( 1.05)   41.0   39 - 42
#  8:    4   21  9 239 (43.35)  237.0 196 - 322
```

Final Table. Eventually you should be able to produce[6] a publication-ready version such as the following table (whose code we are hiding here for compactness, full details are of course in the sources):

Diet	Time	N	Mean_SD	Median	Range
1	0	20	41 (0.99)	41.0	39 - 43
1	21	16	178 (58.70)	166.0	96 - 305
2	0	10	41 (1.49)	40.5	39 - 43
2	21	10	215 (78.14)	212.5	74 - 331
3	0	10	41 (1.03)	41.0	39 - 42
3	21	10	270 (71.62)	281.0	147 - 373
4	0	10	41 (1.05)	41.0	39 - 42
4	21	9	239 (43.35)	237.0	196 - 322

Interpretation. This summary table offers the same interpretation as before, namely that diet 3 has the highest mean and median weights at day 21 but a higher variation than group 4. However it should be noted that at day 21, diet 1 lost 4 chicks from 20 that started and diet 4 lost 1 from 10. This could be a sign of some issues (e.g. safety).

[6] Using the `kable()` function from the `knitr` package with functions from the `kableExtra` package.

Limitations of data. Information on bias reduction measures is not given and is not available either[7]. We don't know if the chicks were fairly and appropriately randomised to the diets and whether the groups are comparible (e.g., same breed of chicks, sex (gender) balance). Hence we should be very cautious with drawing conclusion and taking actions with this data.

15. Conclusion

This "Getting Started in R" guide introduced you to some of the basic concepts underlying R and used a real life dataset to produce some graphs and summary statistics. It is only a flavour of what R can do but hopefully you have seen some of power of R and its potential.

What next. There are plenty of R courses, books and on-line resources that you can learn from. It is hard to recommend any in particular as it depends on how you learn best. Find things that work for you (paying attention to the quality) and don't be afraid to make mistakes or ask questions. Most importantly have fun.

[7] I contacted the source authors and kindly received the following reply *"They were mainly undergraduate projects, final-year, rather than theses, so, unfortunately, it's unlikely that any record remains, particularly after so many years."*