Getting Started in R: Tinyverse Edition

Saghir Bashir and Dirk Eddelbuettel

This version was compiled on November 24, 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 | Tinyverse

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]^.

This note is a variant of the original document[3] but stresses the use of Base R along with careful dependency management as discussed below.

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. https://eddelbuettel.github.io/gsir-te/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.

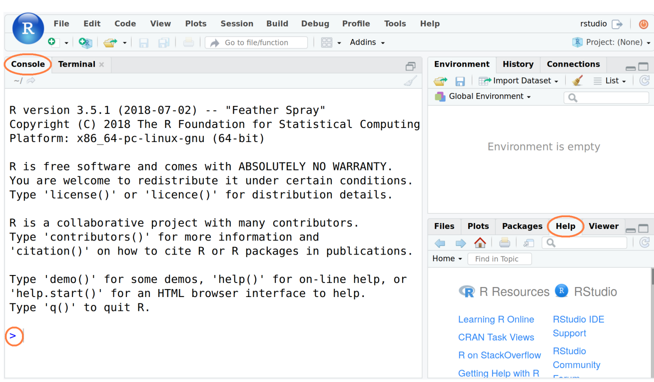


Fig. 1. RStudio Screenshot with Console on the left and Help tab in the bottom right

[1] R project: https://www.r-project.org/
[2] RStudio IDE: https://www.rstudio.com/products/RStudio/
[3] Getting Started with R: https://github.com/saghirb/Getting-Started-in-R

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

We 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)
#
#   mean> x <- c(0:10, 50)
#
#   mean> xm <- mean(x)
#
#   mean> 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.

Exercise. Try the following:

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

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).

Hint. To recall a previously typed commands use the up arrow key (↑). To go between previously typed commands use the up and down arrow (↓) keys. To modify or correct a command use the left (←) and right arrow (→) keys.

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"
```

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] 1.414214
# 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 object.

```
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")
```

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

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

```r
# 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  2.0
#  [10]  2.5  3.0
```

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

```r
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,

```r
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 (deletes) them.

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

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

```r
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:

```r
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:

```r
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 7.0
#  [12] 7.2 7.4 7.6 7.8 8.0
```

Custom Functions. You can create your own functions (using the `function()` keyword) 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 how to do so but for now you don't need to.

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

Extending 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. You want to think carefully before adding dependencies.

The `tinyverse` View. The philosophy of *less is more* is at the core of the tinyverse[4]. Fewer dependencies means a smaller footprint, faster installation, and most importantly fewer nodes in your dependency graph. Experience, as well as empirical and theoretical 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* a complex topic, and there are no easy answers. But by adding another package, we always open a door to interface changes we no longer control. The added functionality 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 view that *fewer is better*, and will rely on only two additional packages: `data.table`[5] for data wrangling as well as input/output, and `ggplot2` [6] for visualization.

Installation. If needed, install these two packages via the following command which should pick the suitable version for your installation:

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

[4] Tinyverse: http://www.tinyverse.org/
[5] data.table: http://r-datatable.com
[6] ggplot2: https://ggplot2.tidyverse.org

8. Chick Weight Data

R comes with many datasets installed[7]. We will use the ChickWeight dataset to learn about data manipulation. The help system gives a basic summary of the experiment from which the data was collect:

> *"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 `fread()` function from the `data.table` package which returns a `data.table` object (whereas the dataset built into R has a different format). 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")
```

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()`[8]. You can also specify a full path, using ~ to denote your home directory.

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
#  ---
#  576:    48    4   18    261
#  577:    48    4   20    303
#  578:    48    4   21    322
```

Several base R functions help us inspect the data: `str()` compactly displays the structure, `summary()` provides a summary, and `head()` and `tail()` display the beginning and end of the data set.

```
str(cw)
#  Classes 'data.table' and 'data.frame':
#    578 obs. of  4 variables:
#  $ Chick : int [1:578] 18 18 16 16 16 ...
#  $ Diet : int [1:578] 1 1 1 1 1 ...
#  $ Time : int [1:578] 0 2 0 2 4 ...
#  $ weight: int [1:578] 39 35 41 45 49 ...
#  - attr(*, ".internal.selfref")=<externalptr>
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. This shows that the dataset has 578 observations and 4 variables as we would expect, and as compared to the original data file `ChickWeight.csv`. So a good start. `str()` call notes the types of variables (all `integer` here) and the first few values. The RStudio 'Environment' pane provides a very similar view.

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

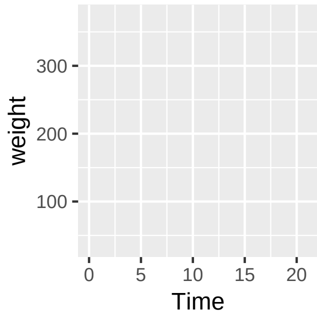
`ggplot2` Package. To visualise the chick weight data, we will use the ggplot2 package. Our interest is in seeing how the *weight changes over time for the chicks by diet*. For the moment don't worry too much about the details just try to build your own understanding and logic. To learn more try different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

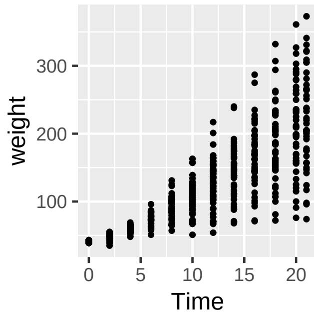
First plot. Let's plot the weight data (vertical axis) over time (horizontal axis).

```
# (Silently) load the plotting package
suppressMessages(library(ggplot2))
# 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()
```

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

[8] Use `getwd()` for the current directory and `setwd("/to/data/path/data.csv")` to change it.

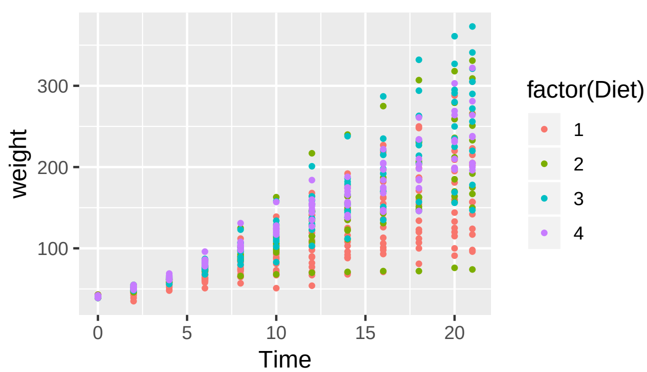




Exercise. Switch the variables `Time` and `weight` in code used for the plot on the right? What do you think of this new plot compared to the original?

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).

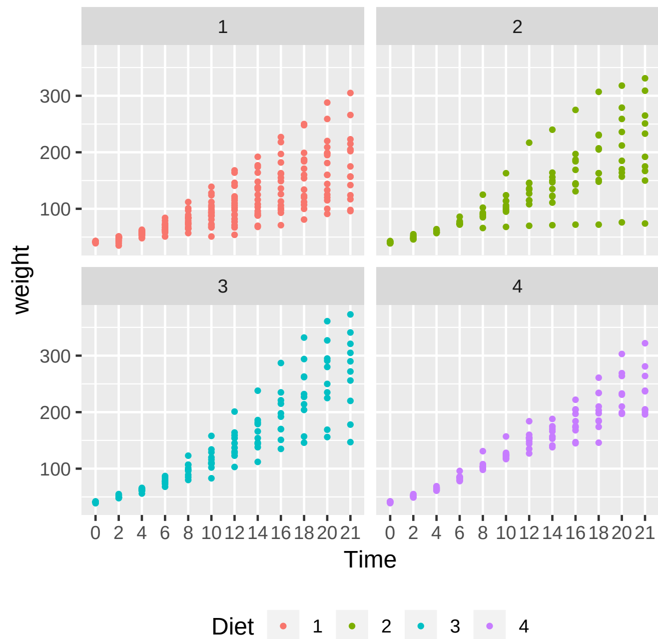
Factor Variables. Before we continue, we have to make an important change to the `cw` dataset by making `Diet` and `Time` *factor variables*. This means that R will treat them as categorical variables instead of continuous variables. It will simplify our coding.

```
cw[, Diet := factor(Diet)]
cw[, Time := factor(Time)]
str(cw)      # notice the difference ?
#  Classes 'data.table' and 'data.frame':
#    578 obs. of  4 variables:
#  $ Chick : int [1:578] 18 18 16 16 16 ...
#  $ Diet : Factor w/ 4 levels "1","2","3","4":
#     1 1 1 1 1 ...
#  $ Time : Factor w/ 12 levels
#     "0","2","4","6",..: 1 2 1 2 3 ...
#  $ weight: int [1:578] 39 35 41 45 49 ...
#  - attr(*, ".internal.selfref")=<externalptr>
```

Notice that the `:=` operator altered the variable "in-place", and no explicit assignment was made. This is a key feature of `data.table` which operated "by reference": changes are made in reference to one instance of the `cw` variable, rather than by creating updated copies. We will revisit this `:=` assignment below.

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

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



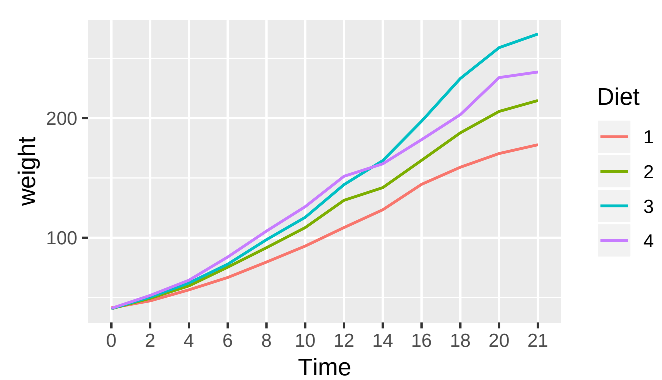
Exercise. To overcome the issue of overlapping points we can ***jitter*** the points using `geom_jitter()`. Replace the `geom_point()` above with `geom_jitter()`. What do you observe?

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

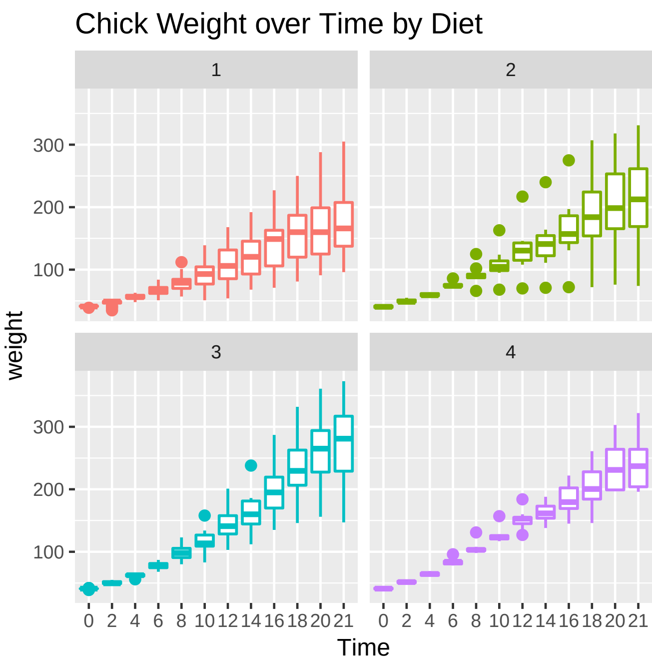


Interpretation. We can see that diet 3 has the highest mean weight gain by the end of the experiment but we don't have any information about the variation (uncertainty) in the data.

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



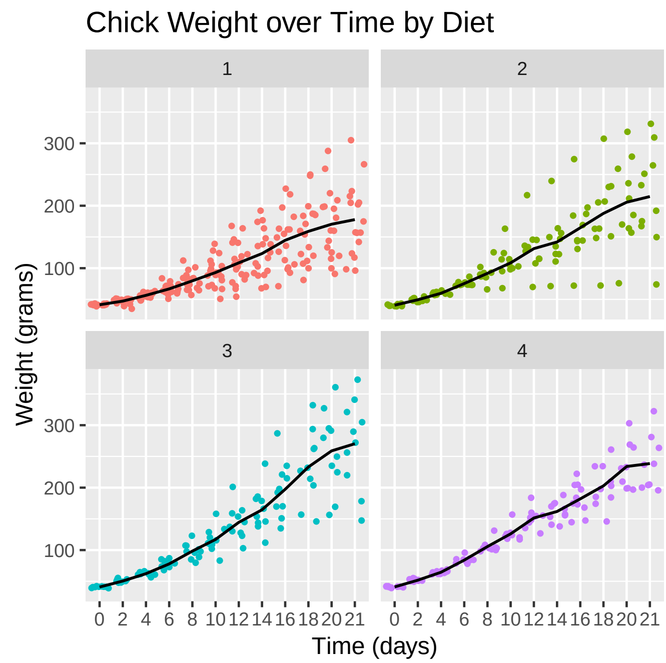
Interpretation. Diet 3 seems to have the highest "average" weight gain but it has more variation than diet 4 which is consistent with our findings so far.

Exercise. Add the following information to the 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(cw, 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. Conceptually, `data.table` operations can be viewed as `dt[i, j, by]` with some intentional similarity to SQL. Here `i` can select (or subset) rows, `j` is used to select, summarise or mutate columns, and `by` is the grouping operator. Numerous examples follow.

`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.

```
cw[, weightKg := weight/1000]    # add a column
cw
#      Chick Diet Time weight weightKg
#   1:    18    1    0     39    0.039
#   2:    18    1    2     35    0.035
#   3:    16    1    0     41    0.041
#  ---
# 576:    48    4   18    261    0.261
# 577:    48    4   20    303    0.303
# 578:    48    4   21    322    0.322
cw[, Diet := paste0("Diet_", Diet)] # mod col.
cw
#      Chick   Diet Time weight weightKg
#   1:    18 Diet_1    0     39    0.039
#   2:    18 Diet_1    2     35    0.035
#   3:    16 Diet_1    0     41    0.041
#  ---
# 576:    48 Diet_4   18    261    0.261
# 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.

```
# Keep variables Time, Diet and weightKg
cw[, .(Chick, Time, Diet, weightKg)]
#        Chick Time   Diet weightKg
#   1:     18    0 Diet_1    0.039
#   2:     18    2 Diet_1    0.035
#   3:     16    0 Diet_1    0.041
#  ---
# 576:     48   18 Diet_4    0.261
# 577:     48   20 Diet_4    0.303
# 578:     48   21 Diet_4    0.322
```

`j` **to summarise.** It can be used to create aggregations, which is particularly handy with the grouping operator. The following example computes means and standard deviations of the 'weight' variable grouped by 'Diet'. Note that the output has been truncated.

```
cw[, .(Mean=mean(weight),SDev=sd(weight)),
   by=.(Diet, Time)]
#       Diet Time     Mean      SDev
#  1: Diet_1    0  41.4000  0.994723
#  2: Diet_1    2  47.2500  4.278157
#  3: Diet_1    4  56.4737  4.128067
# ---
# 46: Diet_4   18 202.9000 33.557413
# 47: Diet_4   20 233.8889 37.568086
# 48: Diet_4   21 238.5556 43.347754
```

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

```
setnames(cw, c("Diet", "weight"),
         c("Group", "Weight"))
cw
#        Chick  Group Time Weight  weightKg
#   1:     18 Diet_1    0     39    0.039
#   2:     18 Diet_1    2     35    0.035
#   3:     16 Diet_1    0     41    0.041
#  ---
# 576:     48 Diet_4   18    261    0.261
# 577:     48 Diet_4   20    303    0.303
# 578:     48 Diet_4   21    322    0.322
```

`i` **operator.** Keeps or drops observations (rows).

```
cw[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. Setting a key changes the order of the observations (rows), and also makes indexing faster.

```
cw[order(Weight)]        # on the fly
#        Chick  Group Time Weight weightKg
#    1:     18 Diet_1    2     35    0.035
#    2:     18 Diet_1    0     39    0.039
#    3:      3 Diet_1    2     39    0.039
#  ---
#  576:     34 Diet_3   21    341    0.341
#  577:     35 Diet_3   20    361    0.361
#  578:     35 Diet_3   21    373    0.373
setkey(cw, Chick, Time)    # setting a key
cw
#        Chick  Group Time Weight weightKg
#    1:      1 Diet_1    0     42    0.042
#    2:      1 Diet_1    2     51    0.051
#    3:      1 Diet_1    4     59    0.059
#  ---
#  576:     50 Diet_4   18    234    0.234
#  577:     50 Diet_4   20    264    0.264
#  578:     50 Diet_4   21    264    0.264
```

Exercise. What does the `order()` do? Try using `order(Time)` and `order(-Time)` in the `i` column.

13. Chaining

You may want to do multiple data wrangling steps at once. This is where the 'chaining' of `data.table` operations (*i.e.*, several sets of commands with square brackets) comes to the rescue:

```
cw21 <- cw[Time %in% c(0,21)][    # i: select rows
  , weight := Weight][            # j: mutate
  , Group := factor(Group)][
  , .(Chick,Group,Time,weight)][  # j: arrange
   order(Chick,Time)][            # i: order
  1:5]                            # i: subset
```

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 summary statistics. We start by looking at the number of observations and the mean of weight grouped by **diet** and **time**.

```
cw[, .(N    = .N,               # .N is nb per group
       Mean = mean(Weight)),    # compute mean
   by=.(Group, Time)][          # group by Diet + Time
   1:5]                         # display rows 1 to 5
#      Group Time  N    Mean
#  1: Diet_1    0 20 41.4000
#  2: Diet_1    2 20 47.2500
#  3: Diet_1    4 19 56.4737
#  4: Diet_1    6 19 66.7895
#  5: Diet_1    8 19 79.6842
```

`by=` **argument.** 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. We can calculate the standard deviation, median, minimum and maximum values—only at days 0 and 21.

```
cws <- cw[Time %in% c(0,21),
          .(N      = .N,
            Mean   = mean(Weight),
            SDev   = sd(Weight),
            Median = median(Weight),
            Min    = min(Weight),
            Max    = max(Weight)  ),
          by=.(Group, Time)]
cws
#      Group Time  N  Mean   SDev Median Min Max
#  1: Diet_1    0 20  41.4  0.995   41.0  39  43
#  2: Diet_1   21 16 177.8 58.702  166.0  96 305
#  3: Diet_2    0 10  40.7  1.494   40.5  39  43
#  4: Diet_2   21 10 214.7 78.138  212.5  74 331
#  5: Diet_3    0 10  40.8  1.033   41.0  39  42
#  6: Diet_3   21 10 270.3 71.623  281.0 147 373
#  7: Diet_4    0 10  41.0  1.054   41.0  39  42
#  8: Diet_4   21  9 238.6 43.348  237.0 196 322
```

Finally, we can make the summaries "prettier" for a possible report or publication where we format the numeric values as text.

```
cws[, Mean_SD := paste0(format(Mean,digits=1),
                        " (",
                        format(SDev,digits=2),
                        ")")]
cws[, Range := paste(Min, "-", Max)]
prettySum <- cws[ , .(Group, Time, N, Mean_SD,
                      Median, Range)][
                 order(Group, Time)]
prettySum
#      Group Time  N     Mean_SD Median     Range
#  1: Diet_1    0 20  41 ( 0.99)   41.0   39 - 43
#  2: Diet_1   21 16 178 (58.70)  166.0  96 - 305
#  3: Diet_2    0 10  41 ( 1.49)   40.5   39 - 43
#  4: Diet_2   21 10 215 (78.14)  212.5  74 - 331
#  5: Diet_3    0 10  41 ( 1.03)   41.0   39 - 42
#  6: Diet_3   21 10 270 (71.62)  281.0 147 - 373
#  7: Diet_4    0 10  41 ( 1.05)   41.0   39 - 42
#  8: Diet_4   21  9 239 (43.35)  237.0 196 - 322
```

Final Table. Eventually you should be able to produce a publication-ready version such as the following table. Its code uses the `kable` and `kableExtra` packages. While the code is not displayed here for compactness, full details are of course in the sources.

Group	Time	N	Mean_SD	Median	Range
Diet_1	0	20	41 (0.99)	41.0	39 - 43
Diet_1	21	16	178 (58.70)	166.0	96 - 305
Diet_2	0	10	41 (1.49)	40.5	39 - 43
Diet_2	21	10	215 (78.14)	212.5	74 - 331
Diet_3	0	10	41 (1.03)	41.0	39 - 42
Diet_3	21	10	270 (71.62)	281.0	147 - 373
Diet_4	0	10	41 (1.05)	41.0	39 - 42
Diet_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).

Limitations of data. Information on bias reduction measures is not given and is not available either[9]. We don't know if the chicks were fairly and appropriately randomised to the diets and whether the groups are comparable (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.

16. Acknowledgements

Special thanks to Saghir Bashir for publishing the initial version of *Getting Started with R*, Brodie Gaslam and Matt Dowle for feedback on this version, and to Joshua Ulrich for many discussions about The Tinyverse.

[9] I (ie Saghir) 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."*

