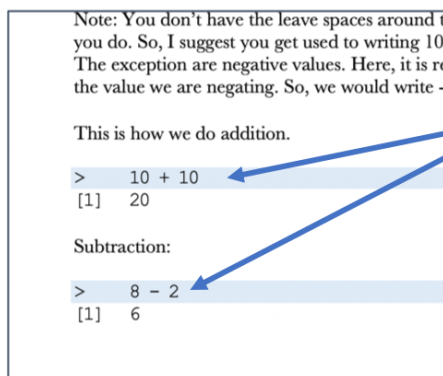


FASS512: Data wrangling and data exploration in R

Professor Patrick Rebuschat, p.rebuschat@lancaster.ac.uk

Please work through the following handout at your own pace.

As in the previous handouts, please type the commands in your computer. That is, **don't just read the commands on the paper, please type every single one of them.**



```
Note: You don't have the leave spaces around t
you do. So, I suggest you get used to writing 10
The exception are negative values. Here, it is re
the value we are negating. So, we would write -

This is how we do addition.
> 10 + 10
[1] 20

Subtraction:
> 8 - 2
[1] 6
```

Every time you see these shaded lines, please **type the commands** either in the console or the script editor, as appropriate.

Finally, this handout assumes that you have completed all previous handouts. If you haven't, please do this before working on the following handout. Handouts are available on [Moodle](#).

References for this handout

Many of the examples and data files from our class come from these excellent textbooks:

- Andrews, M. (2021). *Doing data science in R*. Sage.
- Crawley, M. J. (2013). *The R book*. Wiley.
- Fogarty, B. J. (2019). *Quantitative social science data with R*. Sage.
- Winter, B. (2019). *Statistics for linguists. An introduction using R*. Routledge.

Are you ready? Then let's start on the next page! 📄

Step 0: *Before you start*

To complete this handout, we first need to install and load the tidyverse and descr packages.

```
install.packages("tidyverse")  
  
library(tidyverse)  
  
install.packages("descr")  
  
library(descr)
```

In addition, please download the following data files from Moodle (see folder for session 3) and place them in your working directory.

- nettle_1999_climate.csv
- language_exams_new.csv
- scores.csv

Step 1: *What is a tibble?*

In the last session, you have learned how to install the `tidyverse` package (Wickham, 2017).

`tidyverse` is a collection of packages that greatly facilitates data handling in R. In our session on data visualization, you will encounter the `ggplot2` package (Wickham, 2016), which is part of `tidyverse`. Today, we will use functions from other important packages of the `tidyverse`, namely `tibble` (Müller & Wickham, 2018), `readr` (CITE), and `dplyr` (Wickham et al., 2018). These are all automatically installed when you install the `tidyverse`.

Before we look at data wrangling, we need to convert our data frame into a tibble.

What is a tibble?

Tibbles are like the data frames but better. For example, they load much faster, which is important when you are dealing with lots of data.

To learn how to explore and handle tibbles, let's convert a data frame into a tibble.

First, let's load the data and create a data frame called `languages`. The data set comes from Winter's (2019) textbook.

```
languages <- read.csv('nettle_1999_climate.csv')
languages
```

	Country	Population	Area	MGS	Langs
1	Algeria	4.41	6.38	6.60	18
2	Angola	4.01	6.10	6.22	42
3	Australia	4.24	6.89	6.00	234
4	Bangladesh	5.07	5.16	7.40	37
5	Benin	3.69	5.05	7.14	52
6	Bolivia	3.88	6.04	6.92	38
7	Botswana	3.13	5.76	4.60	27
8	Brazil	5.19	6.93	9.71	209
9	Burkina Faso	3.97	5.44	5.17	75
10	CAR	3.50	5.79	8.08	94
11	Cambodia	3.93	5.26	8.44	18
12	Cameroon	4.09	5.68	9.17	275
13	Chad	3.76	6.11	4.00	126
14	Colombia	4.53	6.06	11.37	79
15	Congo	3.37	5.53	9.60	60
16	Costa Rica	3.49	4.71	8.92	10
17	Cote d'Ivoire	4.10	5.51	8.67	75
18	Cuba	4.03	5.04	7.46	1
19	Ecuador	4.04	5.45	8.14	22
20	Egypt	4.74	6.00	0.89	11
21	Ethiopia	4.73	6.09	7.28	112
22	French Guiana	2.01	4.95	10.40	11
23	Gabon	3.08	5.43	8.79	40
24	Ghana	4.19	5.38	8.79	73
25	Guatemala	3.98	5.04	9.31	52

26	Guinea	3.77	5.39	7.38	29
27	Guyana	2.90	5.33	12.00	14
28	Honduras	3.72	5.05	8.54	9
29	India	5.93	6.52	5.32	405
30	Indonesia	5.27	6.28	10.67	701
31	Kenya	4.41	5.76	7.26	58
32	Laos	3.63	5.37	7.14	93
33	Liberia	3.43	5.05	10.62	34
34	Libya	3.67	6.25	2.43	13
35	Madagascar	4.06	5.77	7.33	4
36	Malawi	3.93	5.07	5.80	14
37	Malaysia	4.26	5.52	11.92	140
38	Mali	3.98	6.09	3.59	31
39	Mauritania	3.31	6.01	0.75	8
40	Mexico	4.94	6.29	5.84	243
41	Mozambique	4.21	5.90	6.07	36
42	Myanmar	4.63	5.83	6.93	105
43	Namibia	3.26	5.92	2.50	21
44	Nepal	4.29	5.15	6.39	102
45	Nicaragua	3.60	5.11	8.13	7
46	Niger	3.90	6.10	2.40	21
47	Nigeria	5.05	5.97	7.00	427
48	Oman	3.19	5.33	0.00	8
49	Panama	3.39	4.88	9.20	13
50	Papua New Guinea	3.58	5.67	10.88	862
51	Paraguay	3.64	5.61	10.25	21
52	Peru	4.34	6.11	2.65	91
53	Philippines	4.80	5.48	10.34	168
54	Saudi Arabia	4.17	6.33	0.40	8
55	Senegal	3.88	5.29	3.58	42
56	Sierra Leone	3.63	4.86	8.22	23
57	Solomon Islands	3.52	4.46	12.00	66
58	Somalia	3.89	5.80	3.00	14
59	South Africa	4.56	6.09	6.05	32
60	Sri Lanka	4.24	4.82	9.59	7
61	Sudan	4.41	6.40	4.02	134
62	Suriname	2.63	5.21	12.00	17
63	Tanzania	4.45	5.98	7.02	131
64	Thailand	4.75	5.71	8.04	82
65	Togo	3.56	4.75	7.91	43
66	UAE	3.21	4.92	0.83	9
67	Uganda	4.29	5.37	10.14	43
68	Vanuatu	2.21	4.09	12.00	111
69	Venezuela	4.31	5.96	7.98	40
70	Vietnam	4.83	5.52	8.80	88
71	Yemen	4.09	5.72	0.00	6
72	Zaire	4.56	6.37	9.44	219
73	Zambia	3.94	5.88	5.43	38
74	Zimbabwe	4.00	5.59	5.29	18

To convert this data frame into a tibble, we can use the `as_tibble()` function.

```
languages <- as_tibble(languages) # Converts the data frame into a tibble
```

If you now type the name of the new tibble, your output should look this.

```
languages
# A tibble: 74 × 5
  Country      Population Area   MGS Langs
  <chr>          <dbl> <dbl> <dbl> <dbl>
1 Algeria         4.41  6.38  6.6    18
2 Angola          4.01  6.1   6.22   42
3 Australia       4.24  6.89  6      234
4 Bangladesh      5.07  5.16  7.4    37
5 Benin           3.69  5.05  7.14   52
6 Bolivia         3.88  6.04  6.92   38
7 Botswana        3.13  5.76  4.6    27
8 Brazil          5.19  6.93  9.71  209
9 Burkina Faso    3.97  5.44  5.17   75
10 CAR            3.5   5.79  8.08   94
# ... with 64 more rows
# ⓘ Use `print(n = ...)` to see more rows
```

As you can see above, the tibble has information about the number of observations (74) and variables (5), the names of the variables (Country, Population, Area, MGS, Langs) and the variable types (character: chr, doubles: dbl, which is a type of numeric vector, and integer: int, also a numeric vector). In addition, the command will display the first ten observations of the variables, lines 1 to 10.

Once we have converted a data frame into a tibble, we can manipulate the data more easily by means of the `dplyr` package (Wickham et al., 2018) in the `tidyverse`. As mentioned, tibbles also load faster than regular data frames.

Step 2: *Data wrangling in the tidyverse*

We will now use tidyverse functions for data wrangling. As discussed in our last session, data wrangling is also referred to as data pre-processing or data cleaning. It simply means preparing your raw data (e.g., the data files from experimental software) for statistical analyses. This entails, for example, dealing with missing values, relabeling variables, changing the variable types, etc.

In this part of the handout, we will look at a few tidyverse function that you can use for data wrangling. For more information, I recommend Chapter 3 of Andrews (2021), which provides a comprehensive introduction to data wrangling using tidyverse.

Let's look at five useful functions for data wrangling with tibbles: `filter`, `select`, `rename`, `mutate`, and `arrange`.

Filter function

The `filter()` function can be used, unsurprisingly, to filter **rows** in your tibble. The `filter()` function takes the input tibble as its first argument. The second argument is then a logical statement that you can use to filter the data as you please.

In the following example, we are reducing the languages tibble to only those rows with countries that have more than 500 languages.

```
filter(languages, Langs > 500)
# A tibble: 2 × 5
  Country      Population Area   MGS Langs
  <chr>          <dbl> <dbl> <dbl> <dbl>
1 Indonesia      5.27  6.28  10.7   701
2 Papua New Guinea 3.58  5.67  10.9   862
```

Or if you are interested in the data from a specific country (say, Angola), you could simply run the following command. This will only display the rows for Angola.

```
filter(languages, Country == 'Angola')
# A tibble: 1 × 5
  Country Population Area   MGS Langs
  <chr>          <dbl> <dbl> <dbl> <dbl>
1 Angola      4.01   6.1  6.22   42
```

Select function

In contrast, you can use the `select()` function to select specific **columns**. To do this, simply add the columns you wish to select, separated by commas, as arguments in the function.

```
select(languages, Langs, Country)
# A tibble: 74 × 2
  Langs Country
  <dbl> <chr>
```

```


1    18 Algeria
2    42 Angola
3   234 Australia
4    37 Bangladesh
5    52 Benin
6    38 Bolivia
7    27 Botswana
8   209 Brazil
9    75 Burkina Faso
10   94 CAR
# ... with 64 more rows
#  Use `print(n = ...)` to see more rows

```

As you might notice, the `select()` function can also be used to change the sequence of the columns. (In the original tibble, Country came first, followed by Langs.)

On the other hand, if you wish to exclude a column, you can do this by using a minus sign in front of the column in question. The command below will select the four columns Country, Population, Area and MGS, but excluded Langs as requested.

```

select(languages, -Langs)
# A tibble: 74 × 4
  Country      Population Area   MGS
  <chr>          <dbl> <dbl> <dbl>
1 Algeria         4.41  6.38  6.6
2 Angola          4.01  6.1   6.22
3 Australia        4.24  6.89  6
4 Bangladesh       5.07  5.16  7.4
5 Benin            3.69  5.05  7.14
6 Bolivia          3.88  6.04  6.92
7 Botswana         3.13  5.76  4.6
8 Brazil           5.19  6.93  9.71
9 Burkina Faso     3.97  5.44  5.17
10 CAR             3.5   5.79  8.08
# ... with 64 more rows
#  Use `print(n = ...)` to see more rows

```

Use the colon operator to select consecutive columns, e.g. the columns from Country to Area, as in the example below.

```

select(languages, Country:Area)
# A tibble: 74 × 3
  Country      Population Area
  <chr>          <dbl> <dbl>
1 Algeria         4.41  6.38
2 Angola          4.01  6.1
3 Australia        4.24  6.89
4 Bangladesh       5.07  5.16
5 Benin            3.69  5.05

```

```

6 Bolivia          3.88  6.04
7 Botswana         3.13  5.76
8 Brazil           5.19  6.93
9 Burkina Faso     3.97  5.44
10 CAR             3.5   5.79
# ... with 64 more rows
# ⓘ Use `print(n = ...)` to see more rows

```

Rename function

A third useful function is called `rename()`. This function can be used to change the name of columns. To do this, you first write the new column (here, `Pop`) followed by an equal sign (`=`) and the old column name (`Population`).

```

languages <- rename(languages, Pop = Population)
languages
# A tibble: 74 × 5
  Country      Pop Area  MGS Langs
  <chr>      <dbl> <dbl> <dbl> <dbl>
1 Algeria    4.41  6.38  6.6    18
2 Angola     4.01  6.1   6.22   42
3 Australia  4.24  6.89  6      234
4 Bangladesh 5.07  5.16  7.4    37
5 Benin      3.69  5.05  7.14   52
6 Bolivia    3.88  6.04  6.92   38
7 Botswana   3.13  5.76  4.6    27
8 Brazil     5.19  6.93  9.71  209
9 Burkina Faso 3.97  5.44  5.17   75
10 CAR       3.5   5.79  8.08   94
# ... with 64 more rows
# ⓘ Use `print(n = ...)` to see more rows

```

Mutate function

The `mutate()` function can be used to change the content of a tibble. For example, you can add an additional column, which in the example below will be the `Langs` column divided by 100.

```

languages <- mutate(languages, Langs100 = Langs/100)
str(languages)
tibble [74 × 6] (S3: tbl_df/tbl/data.frame)
 $ Country : chr [1:74] "Algeria" "Angola" "Australia" "Bangladesh" ...
 $ Pop     : num [1:74] 4.41 4.01 4.24 5.07 3.69 3.88 3.13 5.19 3.97 3.5
 ...
 $ Area    : num [1:74] 6.38 6.1 6.89 5.16 5.05 6.04 5.76 6.93 5.44 5.79
 ...
 $ MGS     : num [1:74] 6.6 6.22 6 7.4 7.14 6.92 4.6 9.71 5.17 8.08 ...
 $ Langs   : num [1:74] 18 42 234 37 52 38 27 209 75 94 ...

```



```
$ Langs100: num [1:74] 0.18 0.42 2.34 0.37 0.52 0.38 0.27 2.09 0.75
0.94 ...
```

Arrange function

Finally, the `arrange()` function can be used to order a tibble in ascending or descending order. In the example below, we use this function to first look at the countries with the smallest number of languages (Cuba, Madagascar, etc.), followed by the countries with the largest numbers of languages (Papua New Guinea, Indonesia, Nigeria, etc.).

```
arrange(languages, Langs) # Ascending order
# A tibble: 74 × 6
  Country      Pop Area  MGS Langs Langs100
  <chr>      <dbl> <dbl> <dbl> <dbl>    <dbl>
1 Cuba        4.03  5.04  7.46     1      0.01
2 Madagascar  4.06  5.77  7.33     4      0.04
3 Yemen       4.09  5.72   0       6      0.06
4 Nicaragua   3.6   5.11  8.13     7      0.07
5 Sri Lanka   4.24  4.82  9.59     7      0.07
6 Mauritania  3.31  6.01  0.75     8      0.08
7 Oman        3.19  5.33   0       8      0.08
8 Saudi Arabia 4.17  6.33  0.4      8      0.08
9 Honduras    3.72  5.05  8.54     9      0.09
10 UAE         3.21  4.92  0.83     9      0.09
# ... with 64 more rows
# ⓘ Use `print(n = ...)` to see more rows
```

```
arrange(languages, desc(Langs)) # Descending order
# A tibble: 74 × 6
  Country      Pop Area  MGS Langs Langs100
  <chr>      <dbl> <dbl> <dbl> <dbl>    <dbl>
1 Papua New Guinea 3.58  5.67 10.9    862      8.62
2 Indonesia        5.27  6.28 10.7    701      7.01
3 Nigeria          5.05  5.97   7     427      4.27
4 India            5.93  6.52  5.32   405      4.05
5 Cameroon         4.09  5.68  9.17   275      2.75
6 Mexico           4.94  6.29  5.84   243      2.43
7 Australia        4.24  6.89   6     234      2.34
8 Zaire            4.56  6.37  9.44   219      2.19
9 Brazil           5.19  6.93  9.71   209      2.09
10 Philippines      4.8   5.48 10.3    168      1.68
# ... with 64 more rows
# ⓘ Use `print(n = ...)` to see more rows
```

Step 3: *Exploratory data analysis*

We are now ready for some exploratory data analysis in R. First, let's load the three data files as tibbles. (Yes, `languages` was already converted to a tibble, so you can skip the first command.)

To load the data sets as tibbles, we use the `read_csv()` function. Note: `read.csv()` imports the data as a data frame, and `read_csv()` imports it as a tibble.

Loading and quickly inspecting the data

```
languages <- read_csv('nettle_1999_climate.csv')
language_exams_new <- read_csv('language_exams_new.csv')
test_scores <- read_csv('scores.csv')
```

Once you have loaded the data sets and created the new tibbles, it's good to inspect the data to make sure all was imported properly. This is important before you do any analyses. Remember: "Garbage in, garbage out."

You can use the `View()` and `str()` functions, for example.

```
View(languages)
View(language_exams_new)
View(test_scores)
```

```
str(languages)
str(language_exams_new)
str(test_scores)
```

We are now ready to calculate a few summary statistics.

Mean

To calculate the arithmetic mean, you can use the `mean()` function.

```
mean(test_scores$scores)
612.2381
```

In the case of `language_exams_new`, we have three exams for which we might want to know the mean.

```
mean(language_exams_new$exam_1)
68.052
```

```
mean(language_exams_new$exam_2)
75.055
```

```
mean(language_exams_new$exam_3)
87.918
```

But rather than calculating the mean for each column separately (`exam_1` to `exam_3`), we can use the `colMeans()` function to calculate the mean for all columns. (Note that for `student_id` the output is essentially meaningless; it might be worth to filter out this variable.)

```
colMeans(language_exams_new)
student_id      age      exam_1      exam_2      exam_3
16326.612      24.705      68.052      75.055      87.918
```

Trimmed mean

The trimmed mean is useful when we want to exclude extreme values. Remember though that any data exclusion needs to be justified and it needs to be described in your report.

Compare the two outputs, with and without the extreme values. The `trim` argument here deletes the bottom and top 10% of scores.

```
mean(test_scores$scores)
612.2381
```

```
mean(test_scores$scores, trim = 0.1)
508.5882
```

Median

The median is calculated by using the following command.

```
median(test_scores$scores)
465
```

```
median(language_exams_new$exam_1)
68
```

Standard deviation

The most widely used measure of dispersion is the standard deviation (SD).

```
sd(test_scores$scores)
552.2874
```

```
sd(language_exams_new$exam_1)
2.049755
```

Range

The range can provide useful information about our sample data. For example, let's calculate the age range of participants in `language_exams_new`.

The `range()` function does not give you the actual range. It only provides the minimum and maximum values.

```
range(language_exams_new$age)
[1] 17 38
```

To calculate the range, you need subtract the maximum value from the minimum, as in the following command.

```
max(language_exams_new$age) - min(language_exams_new$age)
[1] 21
```

Quantiles

Quantiles can easily be displayed by means of the `quantile()` function, as you can see in the example below.

```
quantile(language_exams_new$exam_1)
0%   25%   50%   75%  100%
62   67   68   69   74
```

In our `exam_1` data, each of the quantiles above tells us how many scores are below the given value. For example, 0% of scores are below 62, meaning that 62 is the lowest score. 50% of scores are below 68 and 50% above 68. (The 50% quantile is the median.) And 100% of scores are below 74, meaning that 74 is the highest score in the data set.

We can confirm that this is true by using the `range()` function, which confirms 62 and 74 as minimum and maximum values.

```
range(language_exams_new$exam_1)
[1] 62 74
```

Sometimes, we want to calculate a specific one, e.g. a percentile. For this, you can add the following arguments to the `quantile()` function. For example, to display the 10% and 90% quantiles, you would add 0.1 and 0.9 respectively, as in the examples below.

```
quantile(language_exams_new$exam_1, 0.1)
10%
65
```

```
quantile(language_exams_new$exam_1, 0.9)
90%
71
```

Summary

The `summary()` function is a fast way to get the key summary statistics.

```
summary(round(language_exams_new))
```

student_id	age	exam_1	exam_2	exam_3
Min.:10011	Min.:17.0	Min.:62.00	Min.:70.00	Min.:81.00
1st Qu.:13028	1st Qu.:20.0	1st Qu.:67.00	1st Qu.:74.00	1st Qu.:87.00
Median:15988	Median:23.0	Median:68.00	Median:75.00	Median:88.00
Mean:16327	Mean:24.7	Mean:68.05	Mean:75.06	Mean:87.92
3rd Qu.:19070	3rd Qu.:28.0	3rd Qu.:69.00	3rd Qu.:76.00	3 rd Qu.:89.00
Max.:24996	Max.:38.0	Max.:74.00	Max.:81.00	Max.:97.00

Frequencies

Sometimes it is helpful to observe the frequencies in our sample data. The `freq()` function is really helpful for this. Note: This function requires the `descr` package, which you should have installed already (see Step 0).

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

The following command will create a frequency table. On the left, you will see the score, in the middle the frequency with which the score occurs, and to the right this is expressed in percentage points. This tells you, for example, that the most frequent score (the mode) is 68, the least frequent score is 62.

```
freq(round(language_exams_new$exam_1), plot = FALSE)
```

```
round(language_exams_new$exam_1)
```

	Frequency	Percent
62	2	0.2
63	13	1.3
64	27	2.7
65	68	6.8
66	109	10.9
67	168	16.8
68	196	19.6
69	175	17.5
70	139	13.9
71	56	5.6
72	32	3.2
73	12	1.2
74	3	0.3
Total	1000	100.0

Frequencies with a twist

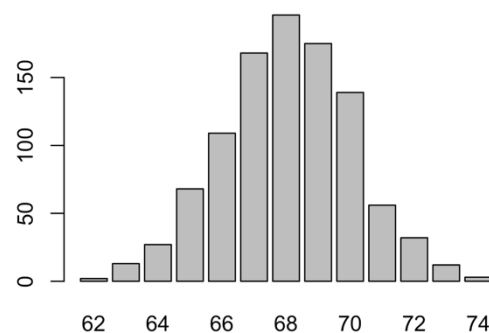
If you omit the argument `plot = FALSE` in the `freq()` function, R will produce both a table and a visual display of your data. Compare the table and the histogram (the graph). Which one is more informative? What are the relative advantages of either?

```
freq(round(language_exams_new$exam_1))
```

```
round(language_exams_new$exam_1)
```

	Frequency	Percent
62	2	0.2
63	13	1.3
64	27	2.7
65	68	6.8
66	109	10.9
67	168	16.8

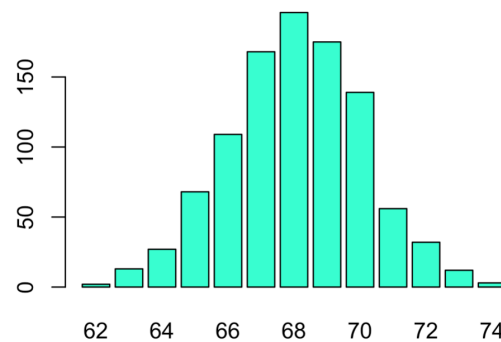
68	196	19.6
69	175	17.5
70	139	13.9
71	56	5.6
72	32	3.2
73	12	1.2
74	3	0.3
Total	1000	100.0



The graphs don't need to be grey, of course. The base R package has an impressive range of available colors. Try out the following, for example.

```
freq(language_exams_new$exam_1, col = 'aquamarine')
```

language_exams_new\$exam_1		
	Frequency	Percent
62	2	0.2
63	13	1.3
64	27	2.7
65	68	6.8
66	109	10.9
67	168	16.8
68	196	19.6
69	175	17.5
70	139	13.9
71	56	5.6
72	32	3.2
73	12	1.2
74	3	0.3
Total	1000	100.0



For more colors, type `colors()`, and see the last pages of the handout, where I display the output of this command.

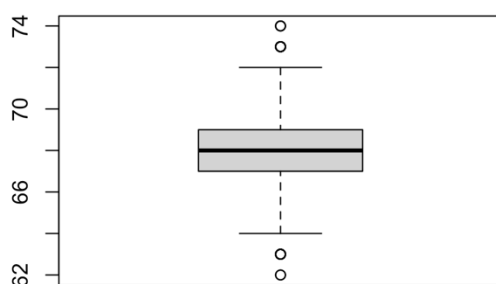
Step 4: *Our first graphic explorations*

To conclude, let's try out a few basic graphics. Next week, we will go into much more detail, but here's a preview of data visualization. The graphics below are available in the R base package.

Boxplot

Boxplots are helpful to inspect the data (Tukey, 1977), as discussed in our lecture today. The following command creates a boxplot, based on `exam_1` from the data set `language_exams_new`.

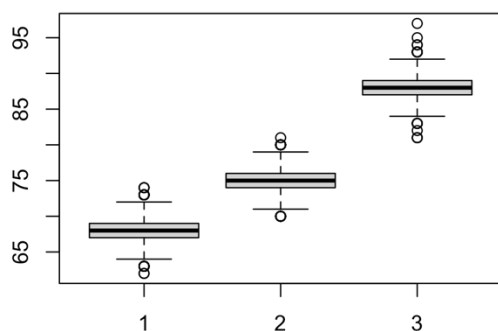
```
boxplot(language_exams_new$exam_1)
```



What if we want to compare performance on all three exams (`exam_1`, `exam_2`, `exam_3`) next to each other?

One way of doing this (there are other ways) is to first create three new objects `exam_1`, `exam_2`, `exam_3`, and then used the `boxplot()` function on the three. Have a go.

```
exam_1 <- c(language_exams_new$exam_1)
exam_2 <- c(language_exams_new$exam_2)
exam_3 <- c(language_exams_new$exam_3)
boxplot(exam_1, exam_2, exam_3)
```



Stem-and-leaf displays

Tukey (1977) also introduce the stem-and-leaf display. For this, let's create first a data frame called `stem_example`, then run the `stem()` function. The items on the left of the vertical bar | are the stem, the items on the right are leaves. Can you figure out how they display the data?

```
stem_example <- c(12, 24, 15, 15, 12, 24, 29, 22, 21, 25, 30, 39, 45,
50, 51)
stem(stem_example)
```

The decimal point is 1 digit(s) to the right of the |

```
1 | 2255
2 | 124459
3 | 09
4 | 5
5 | 01
```

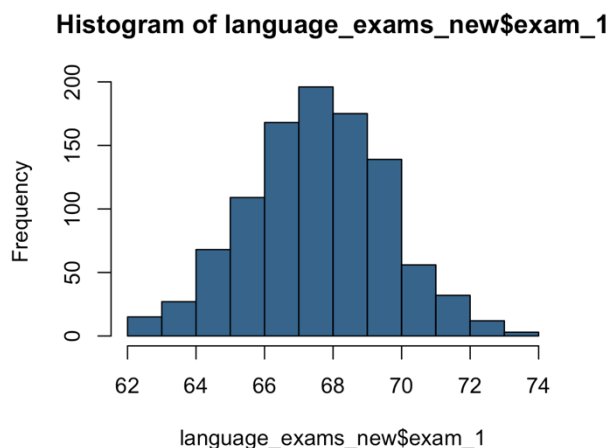
Histograms

Last but not least, histograms are helpful ways to visually inspect your data. To create a histogram, simply use the `hist()` function, as in the following examples.

```
hist(test_scores$scores)
```




```
hist(language_exams_new$exam_1, col = "steelblue4")
```



Just for your information. The `colors()` function will display the colors readily available for your graphics. I placed the output here as it's a useful reference in case you want to try out colors. 😊

```
colors()
```

[1] "white"	[17] "azure4"	[33] "brown1"
[2] "aliceblue"	[18] "beige"	[34] "brown2"
[3] "antiquewhite"	[19] "bisque"	[35] "brown3"
[4] "antiquewhite1"	[20] "bisque1"	[36] "brown4"
[5] "antiquewhite2"	[21] "bisque2"	[37] "burlywood"
[6] "antiquewhite3"	[22] "bisque3"	[38] "burlywood1"
[7] "antiquewhite4"	[23] "bisque4"	[39] "burlywood2"
[8] "aquamarine"	[24] "black"	[40] "burlywood3"
[9] "aquamarine1"	[25] "blanchedalmond"	[41] "burlywood4"
[10] "aquamarine2"	[26] "blue"	[42] "cadetblue"
[11] "aquamarine3"	[27] "blue1"	[43] "cadetblue1"
[12] "aquamarine4"	[28] "blue2"	[44] "cadetblue2"
[13] "azure"	[29] "blue3"	[45] "cadetblue3"
[14] "azure1"	[30] "blue4"	[46] "cadetblue4"
[15] "azure2"	[31] "blueviolet"	[47] "chartreuse"
[16] "azure3"	[32] "brown"	[48] "chartreuse1"