

13: Strings and things

Graphical Analysis of Biological Data

This web page is available as a [PDF file](#)

Strings

Carefully read [R4ds Chapter 14: Strings](#) sections 1-4. I suggest you run the examples but I am not requiring that you do so. The assignment will require you to use some of the functions.

Strings are sequences of characters in `char` format. Any character, including numbers, can be treated as strings. Strings in R are enclosed in double or single quotes. Use double quotes to ensure best practice.

Focus, I mean really **focus** on Sections 3 and 4, regular expressions (regex). Regex is tricky to learn but once you learn it, you are able to harness a lot of power.

For the DNA section, you will have to use the `read_lines()` function to import the file. `read_lines()` reads in the entire file, one line at a time. Each line is treated as a separate string.

The joy of regular expressions

Fair warning: I suspect this section will give you the most trouble. You will use several `stringr` functions in combination with regex patterns. Here are functions you will use at various places in the assignment.

- `str_c()`
- `str_replace()`
- `str_replace_all()`
- `str_to_upper()`
- `str_count()`
- `str_length()`
- `str_extract_all()`
- `str_which()`

Here are some things to remember:

- Read the sections listed above and practice the examples and questions. They are not required but regex patterns take practice.
- R uses backslashes to escape certain characters such as `\t` for tab and `\n` for new line characters. To use these as part of regex pattern strings, you have to escape the backslash with a second backslash, like `"\\t"` or `"\\n"`.
- Use parentheses to capture groups that can be used later. For example, `"(CG)\\1\\1"` would find a repeating pattern of "CGCGCG". This technique is very useful when using `str_replace` when you want to keep part of the original string. Consider this example. You want to remove the letters and the dash, keeping only the numbers.

```
tb <- data.frame(x = c("aa11-bb22", "xx88-yy99"))
tb
```

```
##           x
## 1 aa11-bb22
## 2 xx88-yy99
```

```
tb <- tb %>%
  mutate(x = str_replace(x, "[a-z]+(\\d+)-[a-z]+(\\d+)", "\\1\\2"))
tb
```

```
##           x
## 1 1122
## 2 8899
```

- Use combinations of matching patterns to build up the pattern you want. For example, assume you have species names in the form of “Genus_species”. You want to replace them as “G. species”. Notice the use `\\.` in the replacement pattern to use an actual period.

```
bacteria <- tibble(species = c("Escherischia_coli", "Bacillus_subtilis"))

bacteria <- bacteria %>%
  mutate(species = str_replace(species, "([A-Z]{1}).+(.)$", "\\1\\. \\2"))

bacteria
```

```
## # A tibble: 2 x 1
##   species
##   <chr>
## 1 E. coli
## 2 B. subtilis
```

- Use `|` in regex patterns as “or.” For example, `"this|that"` will match “this” or “that.”

```
strings = c("String with this.", "String with that.", "String with the other.")

str_extract(strings, ".*(this|that).*")
```

```
## [1] "String with this." "String with that." NA
```

- Use `str_count()` to count the number of times a pattern occurs. For example, how many times does a sequence of 2-4 As occur in this DNA sequence? Do not count overlapping sequences. Notice the use of `{...}` to find strings of minimum and maximum lengths. The text for variations of the `{min,max}` pattern.

```
dna <- "ATCACTAAATATGATTTGTGTAAAACCAAAATAAGATCTACAAACGAATAGAAGCTAGAGCGAAAAATGG"

str_count(dna, "[CGT]A{2,4}[CGT]")
```

```
## [1] 6
```

- You can use `\s` or `[:blank:]` to represent whitespace. You can use `\d`, `[:digits:]`, or `[0-9]+` to represent a string of digits. They must be properly formatted as strings, as described in the text.
- Run `example(str_c)` or `?str_c` for an example of how to combine separate strings into a single string. I recommend trying the first one. `example(function_name)` runs the examples shown when you enter `?function_name`.

Separating

Read [R4ds Chapter 12.4.1: Separate](#) for details of separating a single column into multiple columns

The `separate` function is another `tidyr` function to help you separate data in a single column into multiple columns. You have to provide the names of the columns to receive the data with the `into` argument and a regex pattern with the `sep` argument so the function knows where to separate the data. This example separates the names of the Song Sparrow subspecies into three columns. The name is split into the columns, using space (" ") as the separation character.

```
sparrows <- tibble(species = c("Melospiza melodia fisherella",
                              "Melospiza melodia cleonensis",
                              "Melospiza melodia heermanni",
                              "Melospiza melodia mailliardi"))
```

```
sparrows
```

```
## # A tibble: 4 x 1
##   species
##   <chr>
## 1 Melospiza melodia fisherella
## 2 Melospiza melodia cleonensis
## 3 Melospiza melodia heermanni
## 4 Melospiza melodia mailliardi
```

```
sparrows <- sparrows %>%
  separate(species,
    into = c("Genus", "species", "subspecies"),
    sep = " ")
```

```
sparrows
```

```
## # A tibble: 4 x 3
##   Genus      species subspecies
##   <chr>    <chr>    <chr>
## 1 Melospiza melodia fisherella
## 2 Melospiza melodia cleonensis
## 3 Melospiza melodia heermanni
## 4 Melospiza melodia mailliardi
```

Here is another example that has flower colors in a column. The data were not coded consistently, so colors may be separated by spaces, commas, or semicolons. Notice the use of `|` for “or”. We’ll also assume that the first color listed is most important to us, so we can drop extra text using the `extra` argument.

```
flowers <- tibble(color = c("red",
                            "red, pink, or white",
                            "white and yellow",
                            "silver; blue; gold"))
```

```
flowers
```

```
## # A tibble: 4 x 1
##   color
##   <chr>
## 1 red
## 2 red, pink, or white
## 3 white and yellow
## 4 silver; blue; gold
```

```
flowers <- flowers %>%
  separate(color,
```

```
into = "main_color",
sep = ",| |;",
extra = "drop")
```

```
flowers
```

```
## # A tibble: 4 x 1
##   main_color
##   <chr>
## 1 red
## 2 red
## 3 white
## 4 silver
```

The examples above are filled with hints to help you be successful with this assignment.

Join together now

- Read [R4ds Chapter 13.4: Mutating joins](#) for details of how to merge two tibbles. You will use an inner join later.

Joins allow you to merge together separate data frames into one data frame. Many types of joins are possible, depending on what you want to accomplish. Look over Jenny Bryan’s [Join cheatsheet](#). She uses a fun example to show the results of different types of joins.

We will use only an `inner_join` so at a minimum, study her example of an `inner_join` in detail.

Communicating with graphics

- Read [R4ds Chapter 28: Graphics for communication](#) for reference.

Most of our assignments have involved data visualization but we have accepted the default output from `ggplot`, with only a few minor changes. The defaults are good while you are exploring your data. When you are ready to present your data to the broader community (scientific or public), more preparation is necessary.

`ggplot` has a lot of power to make nearly any type of graph you can think of. The depth provided by `ggplot` is greater than we have time to explore but there are a few things I want to touch on.

For starters, you might have wondered why `ggplot` defaults to a gray background for its plots. This relates to what typographers call [typographic color](#) or “page color”, which represents the average look of the text on the page. Most pages you read are black text on whitish paper. The average typographic color is going to be somewhat gray.

This figure shows one of the pages from Zuur et al. that we used earlier. The page on the right was blurred to bring out the typographic color. Notice how the graph in the right panel tends to match the overall “grayness” of the page but the graph in the left panel leaves a big, white hole in the overall grayness.

points beyond these limits are labelled as outliers. Some researchers routinely then remove these observations. Figure 3a shows an example of such a graph using 120 observations of a morphometric variable wing length of the albatross species *Diomedea exulans* (Cipriotti, Egehall & Røhr 2000). The graph has one or two outliers (perhaps wrongly, as we will see in a moment) that there are seven outliers.

Another very useful, but highly neglected, graphical tool to visualize outliers is the Cleveland display (Cleveland 1983). This is a graph in which the raw number of an observation is plotted vs. the observation value, mostly providing much more detailed information than a boxplot. Points that stick out on the right-hand side, or on the left-hand side, are observed values that are considerably larger, or smaller, than the majority of the observations, and require further investigation. If such observations exist, it is important to check the raw data for errors and assess whether the observed values are reasonable. Figure 3b shows a Cleveland display for the species wing length data, note that the observations identified by the boxplot are not especially extreme after all. The y-axis of a Cleveland display is the raw number of an observation, not the weight. There is one observation of a wing length of about 80 mm that stands out in the left about half way up the graph. This value is not considerably larger than the other values, so we cannot say per se that it is an outlier.

Figure 3 shows a multi-panel Cleveland display for all of the morphometric variables measured, note that some variables have a few relatively large values. Such extreme values could indicate true measurement error (e.g. some of the characteristics of 'ultrafast' distastefulness) some Morgan 2006, whereby the observed type is chosen in the wrong number as a measurement scale. Note that one should not try to argue that such large values could have occurred by chance. If they were, then intermediate values should also have been generated by

chance, but none were. (A useful exercise is to generate, separately, an equivalent number of random observations from an appropriate distribution, e.g. the Normal distribution, and determine how the number of extreme-point compares to the empirical data.) When the most likely explanation is that the extreme observations are measurement (observed) errors, they should be dropped because their presence is likely to distort the analysis. For example, we applied a dominance analysis on the full species data set to see whether observations did indeed change observations, and found that for the first two years were mostly determined by the outliers.

So far, we have looked at outliers as outliers as observations that stick out from the rest. A more rigorous approach is to consider whether unusual observations are truly influential on an analysis (e.g. on estimated parameters). We make a distinction between influential observations in the response variable and in the covariates. An example of the latter is when species abundances are modelled as a function of temperature, with mostly all temperature values between 15 and 20 °C, but one of 25 °C. In general, this is not an ideal sampling design because the range 20–25 °C is inadequately sampled. In a field study, however, there may have been only one opportunity to sample the higher temperature. With a large sample size, such observations may be dropped, but with relative small data sets the consequent reduction in sample size may be undesirable, especially if other observations have suffered in other important variables. If omitting such observations is not an option, the alternative is transforming the explanatory variables.

In regression-type models, outliers in the response variable are more complicated to deal with. Transforming the data is an option, but as the response variable is of primary interest, it is better to choose a statistical method that uses a probability distribution that allows greater variation for large mean values (e.g. gamma for continuous data, Poisson or negative binomial for count data) because doing this allows us to

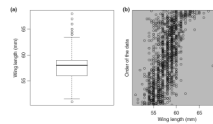


Fig. 3. (a) Multi-panel Cleveland display for all of the morphometric variables measured. The y-axis is the raw number of an observation, and the lower and upper ends of the box are the 25th and 75th quantiles respectively. The lines indicate 1.5 times the size of the box, which the 75th quantile 25% quantile. This is the boxplot of the data. (b) Multi-panel Cleveland display for the same data, but with a y-axis from 0 to 100 and an x-axis from 0 to 80. The y-axis is the raw number of an observation, and the lower and upper ends of the box are the 25th and 75th quantiles respectively. The lines indicate 1.5 times the size of the box, which the 75th quantile 25% quantile. This is the boxplot of the data. (c) Multi-panel Cleveland display for the same data, but with a y-axis from 0 to 100 and an x-axis from 0 to 80. The y-axis is the raw number of an observation, and the lower and upper ends of the box are the 25th and 75th quantiles respectively. The lines indicate 1.5 times the size of the box, which the 75th quantile 25% quantile. This is the boxplot of the data.

© 2009 The Authors. Journal compilation © 2009 British Ecological Society, *Methods in Ecology and Evolution*, 1, 3–14

Studies by typographers suggest that pages with more uniform color are easier to read and cause less straining. That is why **ggplot2** defaults to a medium gray background. This is also one default that many people prefer to change for the final output, perhaps in part because we're used to graphs with white background.

ggplot2 can be customized with little to lots of effort, depending on what you want to accomplish. **ggplot2** also comes with several **built-in “themes”** that you can add to a plot to quickly change the appearance of the plots. I will often use **theme_minimal()**, **theme_bw()**, or **theme_classic**.

Additional themes are available via the **ggtthemes** package. I show an example with **theme_tufte()** in this assignment. We'll use **theme_map()** next assignment I encourage you to try different themes. Eventually, you will probably dig more deeply under the theme hood to tweak your plots even further.

The other thing I want to mention is color. Color is not always necessary. If you can **get it right in black and white**, then your graph has the power to convey detailed information efficiently. However, color is often helpful and it is easier than ever to generate and publish color figures. But, different colors do *not* convey information equally.

Here are some references for you to reference and think about visualizing your data in the future.

[Expert Color Choices for Presenting Data](#) by Maureen Stone.

[Practical Rules for Using Color in Charts](#) by Stephen Few.

[Uses and Misuses of Color](#) by Stephen Few.

[What about “color blindness?”](#) by Maureen Stone.

Some of the referenced articles discuss colorblindness. Colorblindness is not an accurate term because most non-blind people see color¹ but not all people see color the same way. Choosing the right combination of colors is critical to present your data fairly and so everyone can interpret your results easily. A good rule of thumb is to not depend on just color to distinguish groups in a graph. Shapes and shades of gray, as you have seen, are also useful and often preferred.

When you prepare figures for others to see, you should run draft versions through a [colorblind simulator](#), which allows you to see how your figure looks to those with other forms of color vision.

¹[Achromatopsia](#) is one form of true color blindness.